ION SMEUREANU MARIAN DÂRDALĂ

ADRIANA REVEIU

VISUAL C# .NET



Lucrarea poate fi comandată la:

- **Editura CISON tel. 0740092349 sau 021/7787193**
- **Catedra de Informatică Economică - ASE, str. Calea Dorobanți, nr. 15-17, et. 4 cam. 2413, tel. 021/2112650 int. 336, 318 sau 0722747786.**

Editura CISON, str. Constantin Caracas, nr. 9, sect. 1, București, 2004

Prezentarea grafică, procesarea și tehnoredactarea textului aparțin în exclusivitate autorilor.

ISBN 973-8301-12-2

ION SMEUREANU MARIAN DÂRDALĂ

ADRIANA REVEIU

VISUAL C# .NET

EDITURA CISON

București 2004

CUPRINS

	Pag.
1. DIFERENȚE ESENȚIALE ÎNTRE LIMBAJELE C# ȘI C++	9
1. Tipuri referențiale și tipuri valorice	9
Șiruri de caractere	11
2. Vectori și colecții	14
Colecții	17
3. Proprietăți	19
Proprietăți statice	20
4. Delegări și evenimente	22
Mecanismul delegării	22
Evenimente	25
5. Interfețe	29
2. IMPLEMENTAREA ÎN C# A STRUCTURILOR DE DATE DINAMICE	34
1. Aspecte generale	34
2. Implementarea structurilor de date dinamice liniare	35
Structura de tip listă	35
Structura de tip stivă	41
3. Implementarea structurilor de date dinamice arborescente	42
3. INTRODUCERE ÎN WINDOWS FORMS	48
1. Crearea unei aplicații Windows	48
Crearea unei aplicații Windows prin scriere de cod sursă	48
Crearea vizuală a unei aplicații Windows	53
Gestiunea unitară a controalelor unei forme	59
2. Derivarea controalelor	69
3. Modificarea dinamică a proprietăților	71
4. Utilizarea indexărilor	74
4. VALIDAREA DATELOR. GESTIUNEA ERORILOR ȘI A EXCEPȚIILOR	83
1. Validarea datelor introduse într-un formular	83
Validare simplă	83
Validări încrucișate	85
2. Gestiunea excepțiilor	87
Blocurile `try` și `catch`	87
Blocul `finally`	91

Instrucțiunea throw 91

5. GESTIUNEA MOUSE -ULUI ŞI A TASTATURII 95
1. Gestiunea evenimentelor generate de mouse 95
2. Gestiunea evenimentelor generate de tastatură 98

6. LUCRUL CU MENIURI ŞI BARE 107
1. Meniu principal - MainMenu 108
2. Meniuri contextuale - ContextMenu 111
3. Controale de tip bară de instrumente – ToolBar 114
4. Controale de tip bară de stare - StatusBar 120

7. CONTROALE COMPLEXE DE VIZUALIZARE 122
1. Vizualizare liniară - controlul ListView 122
Mecanismul de lucru cu ListView 122
Abordarea programatică a controlului ListView 126
2. Vizualizare arborescentă – controlul TreeView 128

8. LUCRU CU FERESTRE MULTIPLE 143
1. Ferestre secundare de dialog 143
2. Aplicații de tip Multiple Document Interface - MDI 148

9. SUPORTUL .NET PENTRU IMPRIMARE ŞI PREVIZUALIZARE 155
1. Clasa PrintDocument 155
2. Previzualizarea documentului de imprimat 161
3. Imprimarea documentelor utilizând generatorul de rapoarte *CrystalReports* 166

10. VIZUALIZĂRI SPLITATE. ELEMENTE DE GRAFICĂ 175
1. Docarea şi ancorarea controalelor. Vizualizări splitate. Panel-uri. 175
Controlul Splitter 175
2. Elemente de grafică 176

11. LUCRU CU CLIPBOARD-UL. OPERAȚIA DE DRAG AND DROP 200
1. Clipboard-ul 200
2. Operația de drag and drop 204
3. Drag and drop pe tipuri definite de programator 208

12. LEGAREA DATELOR DE INTERFAȚA UTILIZATOR - DATA BINDING 219
1.Legarea simplă – *simple data binding* 219
2.Legarea complexă - *complex data binding* 223

13. ADO.NET - OBIECTELE DE LUCRU CU BAZE DE DATE 227
1. Introducere în tehnologia ADO.NET 227
2. Conexiunea la baza de date 229
3. DataAdapter - adaptorul 232
4. DataSet – mulțimea de date 236
5. Tabela de date – DataTable 239
6. Relațiile dintre date – obiectul DataRelation 241
7. DataRow – tuplu de date 248
8. DataView – vizualizarea 251
9. Obiecte de tip Command 255
10. Lucru cu procedurile stocate 259
11. Typed şi Untyped DataSet 263

14. APLICAȚII CU BAZE DE DATE SUB ADO.NET 267
1. Legarea câmpurilor dintr-o tabelă a bazei de date cu proprietățile unor controale din macheta unei aplicații 267
2. Abordarea programatică a lucrului cu baze de date sub ADO.NET 275
3. Abordarea vizuală a lucrului cu baze de date 283

15. CONTROALE DE UTILIZATOR 291
1. Construirea şi testarea controalelor de utilizator 291
2. Construirea bibliotecilor de componente utilizator 298
Componente fără interfață vizuală 298
Componente cu interfață vizuală 302
INDEX 308
BIBLIOGRAFIE 311

DIFERENȚE ESENȚIALE ÎNTRE LIMBAJELE C# ȘI C++

1. **Tipuri referențiale și tipuri valorice**
2. **Vectori și colecții**
3. **Proprietăți**
4. **Delegări și evenimente**
5. **Interfețe**

1. Tipuri referențiale și tipuri valorice

Acest capitol urmărește punctarea principalelor diferențe între limbajele C# și C++. Testarea celor discutate s-a făcut pe aplicații de tip Console Application, pentru a nu complica înțelegerea cu elemente noi de interfață grafică.

În C# nu mai pot fi definite metode și variabile la nivel global, ci doar în clase și structuri; problema inițializării variabilelor statice în afara clasei s-a rezolvat folosind **constructori declarați static**; ei nu instanțiază obiecte, ci doar alocă și inițializează membrii statici.

Funcțiile statice pot fi apelate și fără a declara obiecte ale clasei.

Tipurile de date folosite în C# se împart în două categorii: ***tipuri valorice*** care sunt manipulate direct, prin numele lor și ***tipuri referențiale***, alocate dinamic și care sunt manipulate prin referințe.

În categoria tipurilor valorice intră ***tipurile de bază*** și cele introduse prin ***struct*** și ***enum***.

În categoria tipurilor referențiale intră tipurile introduse cu ***class***, ***interface*** și ***delegate***.

Programul de mai jos surprinde diferențele dintre cele două categorii:

```
class Class1
 {
      public int Val = 0;
 }
struct Struct1
 {
      public int Val;
 }

class Test
{
      static void Main()
      {
```

```
        Struct1 tv1; tv1.Val = 0;
        Struct1 tv2 = tv1;      tv2.Val = 123;

        Class1 tr1 = new Class1();
        Class1 tr2 = tr1;       tr2.Val = 123;

        Console.WriteLine("Valoric:     {0}, {1}",
                                tv1.Val, tv2.Val);
        Console.WriteLine("Referential: {0}, {1}",
                                tr1.Val, tr2.Val);
        Console.Read();
    }
}
```

Și structura (**tv** - tip valoric) și clasa (**tr** - tip referențial) au câte un singur membru: **Val**; pentru ambele categorii se face același lucru: se definesc câte două instanțe, a doua fiind inițializată cu prima, după care a doua este modificată printr-o atribuire.

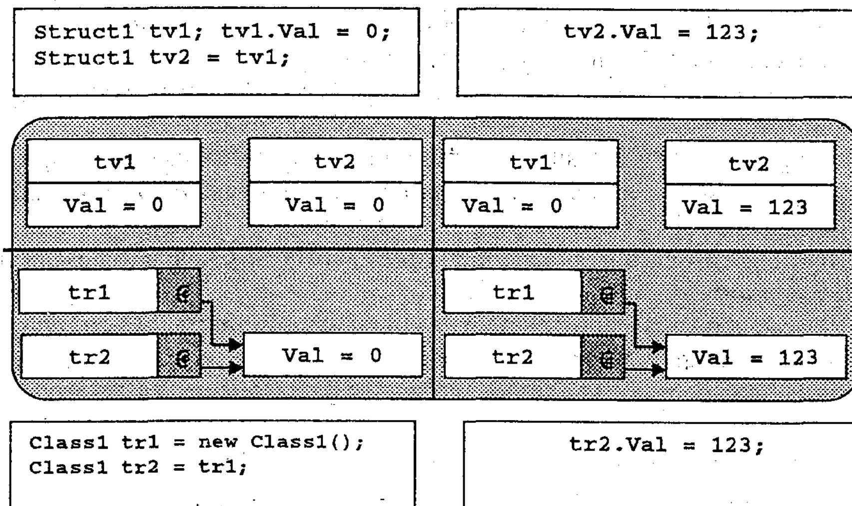


Fig. 1.1 Comportamentul tipurilor valorice și al tipurilor referențiale

Ieșirea afișată indică totuși diferențe majore:

```
Valoric:     0, 123
Referential: 123, 123
```


confirmând că tipurile referențiale sunt manipulate prin referință. Cheia explicației stă în atribuire: pentru tipurile referențiale, atribuirea se face între referințe, **tr2** referind după atribuire de fapt tot primul obiect, ca și **tr1**, fără a avea de-a face cu copieri de obiecte.

Tipurile valorice fiind manipulate direct, prin numele lor, atribuirea presupune copierea conținutului din membrul drept, peste cel stâng.

Spre deosebire de C++ unde o referință trebuia obligatoriu încărcată la declarare, mai târziu nemaiputându-se modifica (referind mereu același obiect), căci operațiile cu referințe se exercitau de fapt asupra obiectelor referite, în C# referințele pot fi definite nule, pot fi încărcate cu obiecte generate cu `new` sau cu referințe de obiecte deja existente. Așadar, în C# conceptul de referință se apropie mai mult de conceptul de pointer, numai că se dereferă automat.

Tipurile referențiale pot fi instanțiate numai cu *new* și deci sunt stocate numai în *heap* (memorie permanentă) , iar tipurile valorice se generează pe stivă (*stack*)

Operatorul *new* se adaptează returnând instanțe pentru tipurile valorice și referințe pentru tipurile referențiale.

Tipurile valorice sunt "structuri" și pot ține metode; chiar și tipurile de bază au metode; spre exemplu tipul `int` (suplinit aici printr-o constantă) are o metodă de conversie în `string`, pentru afișare:

```
string txt = 110.ToString();
```

Există **tipul decimal**, pe 16B, care asigură o precizie mărită (**1 e-28, 7.9 E+28**), fiind folosit pentru calcule monetare; constantele decimal se dau terminând numărul cu m sau M (de la *money*):

```
decimal sold = 123456789.12345m;
```

Două expresii de tip `object` sunt considerate egale dacă ambele referă același obiect, sau dacă ambele sunt `null`.

Șiruri de caractere

C# dispune de tipul **string**; string este un alias pentru clasa `System.String`; ca tipologie se încadrează în tipuri referențiale, dar cu unele particularități. Clasa `String` este derivată direct din `Object` și este o clasă *sealed*, adică din ea nu se mai poate deriva altă clasă. Clasa `String`

este *immutable*, adică un obiect odată creat nu mai poate fi modificat; eventuale modificări conduc la obținerea altui obiect, care va fi adresat folosind aceeași referință; șirurile rămase nereferite sunt colectate de *garbage collector* în vederea dezalocării lor ulterioare.

Clasa **string** dispune de metode precum `Compare`, `Concat`, `Length`, `ToUpper`, `ToLower`, `Substring` etc.

Exemple de lucru cu șiruri:

```
string s1 = "text";
string text = "linia 1 \r\n linia 2";
Console.WriteLine(text);
string fis1 = "c:\\stud\\CS\\siruri\\fisier.txt";
string fis2 = @"c:\stud\CS\siruri\fisier.txt";
                // copie la indigo
char c = s1[0];
s1 += text;
string x = "sir", y = "sir", z; z = x;
if(x == y) Console.WriteLine("Acelasi");
if(x == z) Console.WriteLine("Acelasi");
```

Pentru a evita apelul de constructor cu **new**, se pot face direct inițializări pornind de la șir clasic de caractere : `string s1 = "text";` inițializarea este explicitată de compilator prin:

```
string s = new string(new char[]{'t','e','x','t'});
```

Secvențele de escape sunt recunoscute și onorate la afișare (exemplu, `\r` - *return* și `\n` - *newline*). Dacă se dorește **netratarea secvențelor de escape, se prefixează constanta cu caracterul** @. Pentru că șirul este tratat exact așa cum l-am redactat, varianta se mai numește "copie la indigo".

Operatorul de indexare [ ] poate fi folosit la localizarea unui caracter din șir. Cocatenarea se poate face și cu operatorul +=.

Două șiruri sunt egale fie când referă același obiect, **fie când au același conținut**, lucru care nu se întâmplă cu alte tipuri de obiecte.

În afara tipurilor de bază, programatorii pot defini noi tipuri valorice prin ***enum*** și ***struct***, și noi tipuri referențiale prin ***class***, ***interface*** și ***delegate***.


```
public enum Color     {  Rosu, Albastru, Verde }
public struct Point {  public int x, y; }

public interface IBaza   {      void F( );}
public interface IDerivata: IBaza    {      void G();    }

public class A
{
       protected virtual void H() { Console.WriteLine("A.H");}
}

public class B: A, IDerivata
{
 public void F()
      { Console.WriteLine("B.F, implementare IDerivata.F"); }
 public void G()
      { Console.WriteLine("B.G, implementare IDerivata.G"); }
 override protected void H()
      {      Console.WriteLine("B.H, override pe A.H");       }
}
public delegate void DelegatVid();
```

Trecerea din valoare în referință și invers se face prin împachetare și despachetare (***boxing*** și ***unboxing***):

```
class Test
{
      static void Main()
      {
            int i = 123;
            object o = i;      // boxing
            int j = (int) o;   // unboxing
      }
}
```

Toate tipurile (inclusiv cele de bază) sunt derivate din **object** și moștenesc o metodă comună, de conversie în string:

```
Console.WriteLine(3.ToString());
```

2. Vectori şi colecţii

Vectorii se declară sub forma: `tip [] nume;` avantaj: dimensiunea nefăcând parte din declaraţie nu trebuie dată drept constantă la compilare, ci se dă abia la execuţie, în funcţie de necesităţi. Programatorii de C++ recunosc în această modalitate de gestiune a masivelor, manipularea prin pointeri (în C#, referinţe) a masivelor alocate dinamic.

```
using System;
class Test
 {
      static void Main(string[] args)
      {
       int [,] mb1 = new int[2,3];   //alocare completa
       mb1[1,2]=10;
       int [,] mb2 = { {10,11}, {20,21}, {30,31} };

       int [][] ms1 = new int[3][];
             // aloca doar referintele de linie

       ms1[0]= new int[3];     // alocare linie 1
       ms1[1]= new int[7];     // alocare linie 2 cu alta dim
       ms1[2]= new int[2];

       int [][] ms2 = new int[3][];
       ms2[1]= new int[7];
       Console.Write("\n"+ms2[1][3]);
       ms2 = ms1; // se copiaza referinte, nu elemente
       ms1[1][3]=13;    Console.Write("\n"+ ms2[1][3]);
       Console.Read();
      }
}
```

Programul exemplifică elementele de bază în lucru cu masive bidimensionale în C#. `mb1` este un masiv bidimensional clasic; dimensiunile lui se dau la alocarea cu `new`, sau sunt deduse din lista de iniţializatori, cum este cazul lui `mb2`. Se observă că indicii nu au paranteze de indexare individuale (`mb1[1,2]` şi nu `mb1[1][2]`).

Există şi declaraţia cu dublă indexare, dar ea corespunde masivelor în scară. `ms1` este un masiv în scară (**jagged**, în **zig-zag**); el necesită la alocare doar precizarea numărului de linii, nu şi dimensiunea lor, căci face alocări doar pentru trei referinţe de linii. Abia ulterior, când e nevoie, se fac alocări pentru fiecare linie în parte, acestea putând avea dimensiuni diferite, de unde şi numele acestui tip de masiv.

La copierea unui masiv se copiază doar referinţa, nu şi conţinutul propriu-zis al masivului; dovadă că la prima afişare se indică valoarea `0` cu

care a fost iniţializat implicit `ms2[1][3]`, iar la a doua afişare valoarea `13`, atribuită de fapt lui `ms1[1][3]=13` !!.

Vectorii de tipuri valorice (tipuri fundamentale, `enum`, `struct`) ţin valori, adică numele vectorului este referinţă, **dar la capătul referinţei sunt valori** (fig. 1.2).

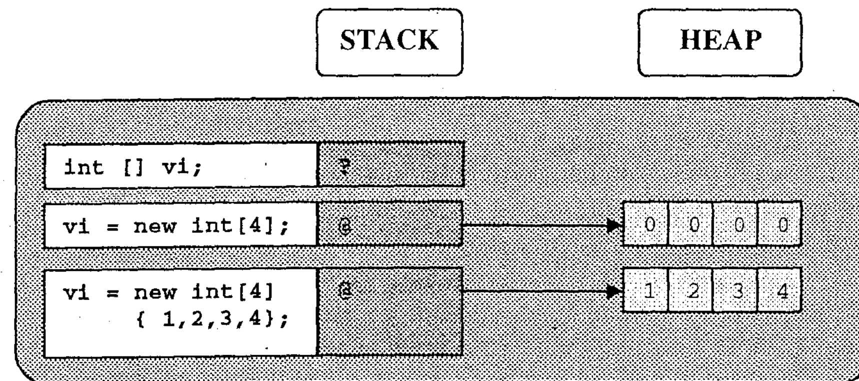


Fig. 1 2: Utilizarea vectorilor de tipuri valorice

Vectorii de obiecte ţin referinţe de obiecte, adică numele vectorului este referinţă, **iar la capătul referinţei sunt alte referinţe**, la zone de memorie statică în care au fost alocate şi instanţiate obiecte (fig. 1.3).

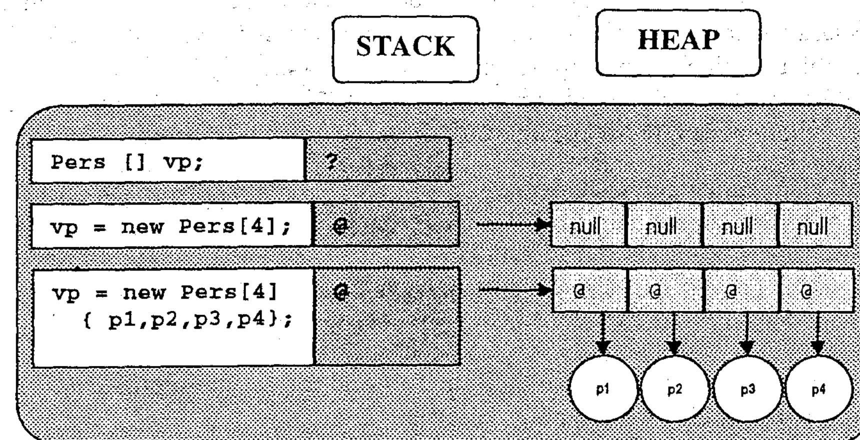


Fig. 1 3. Utilizarea vectorilor de obiecte (tipuri referenţiale)

```
class Pers
    {
        public int Marca;
        public string Nume;
        public Pers(int m,string n){ Marca=m; Nume=n;}
    }

class Principal
    {
        static void Main(string[] args)
        {
            Pers p1;     p1 = new Pers(1,"Unu");
            Pers [] vp; vp = new Pers[4];
            vp[0]= new Pers(0,"Zero"); vp[1]=p1;
            Console.Write("{0} - {1} \n\n",
                vp[1].Marca, vp[1].Nume);
            vp[2]= new Pers(2,"Doi");
            foreach ( Pers crt in vp)
                Console.Write("{0} - {1} \n",
                        crt.Marca, crt.Nume);
            Console.Read();
        }
    }
```

În C#, declarația `Pers p1;` nu înseamnă ca în C++ și generarea unei instanțe, ci declară o referință nulă, ce trebuie încărcată ulterior sau la definire cu adresa unei instanțe deja existente sau nou generată cu `new` :

```
Pers p1, p2; p1 = new Pers(1,"Unu"); p2 = p1;
```

Similar, declarația vectorului de obiecte `Pers [] vp;` înseamnă declararea unei variabile de tip vector de obiecte și nu va instanția nimic; variabila în sine se instanțiază cu `vp = new Pers[3];` moment când i se precizează dimensiunea în funcție de câte referințe va stoca; după aceasta, elementele referință se încarcă cu adrese de obiecte existente sau create adhoc.

```
vp[0]= new Pers(0,"Zero"); vp[1]=p1;

int[] vi = {1,2,3,4};
```

este doar o prescurtare ce va fi explicitată de compilator prin

```
int[] vi = new int[4]{1,2,3,4};
```

Vectorii pot fi parcurși clasic, folosind *for* și adresare indexată, sau pot fi parcurși cu *foreach*, ca și colecțiile.

```
foreach ( Pers crt in vp)
    Console.Write("{0}- {1} \n",crt.Marca, crt.Nume);
```


În nici una din extensiile următoare, ale programului de mai sus, nu se crează noi obiecte, ci doar referințe, una pentru tot vectorul de mai sus:

```
Pers [] copie = vp; // o referinta la intreg masivul
foreach ( Pers crt in copie)
    Console.Write("{0} - {1} \n",crt.Marca, crt.Nume);
```

sau câte una pentru fiecare din obiectele pointate de vectorul inițial:

```
copie = new Pers[vp.Length]; // referinte la fiecare obiect
for(int i=0; i<vp.Length; i++)       copie[i]= vp[i];
foreach ( Pers crt in copie)
    Console.Write("{0} - {1} \n",crt.Marca, crt.Nume);
```

Programul de mai sus funcționează fără modificări, dacă punem ***struct*** în loc de ***class***, dar trebuie avut în vedere că în acest caz atribuirile înseamnă copieri de instanțe, nu comutări de referință.

Vectorii sunt derivați din clasa **`Array`** și moștenesc metode precum **`CopyTo`**, sau proprietăți, precum **`Length`**.

Colecții

Un dezavantaj major al **vectorilor este că au dimensiune fixă**; chiar dacă se dă la momentul execuției și nu la compilare, dimensiunea odată declarată, rămâne fixă; dacă avem nevoie de un vector cu mai multe elemente, trebuie să alocăm un alt vector cu o dimensiune mai mare și să copiem noi înșine, element cu element, vechiul vector la noua locație.

Deși țin tot elemente de același fel (omogene) ca și vectorii, colecțiile se alocă element cu element, fiind redimensionabile dinamic.

O altă deosebire esențială dintre colecții și vectori este că **elementele unui vector au tip, pe când elementele unei colecții nu;** ele sunt de tip generic `object`; cum orice tip (inclusiv cele fundamentale) sunt derivate din `object` și se pot converti la tipul de bază, rezultă că o colecție poate ține orice tip de date, cu condiția ca toate elementele să fie de același tip.

Se rezolvă implicit în acest fel și problema templatizării, în sensul că se poate defini și lucra cu o colecție de tip generic, precizarea tipului făcându-se la momentul folosirii datelor din colecție.

Chiar aplicate pe tipurile fundamentale, colecțiile țin tot obiecte, nu valori ca vectorii; așadar când țin tipuri valorice se aplică procedurile de

boxing/unboxing, prin care un tip valoric e transformat în obiect, adică din tip valoric în tip referenţial şi invers.

Ca asemănări, putem menţiona faptul că atât vectorii, cât şi colecţiile permit adresarea indexată:

```
lista[1]=lista[2];
vect[1]=vect[2];
```

Cele mai utilizate colecţii sunt *ArrayList* (lista) , *SortedList* (map), *Queue* (coadă), *Stack* (stiva) şi necesită includerea namespace-ului adecvat:

```
using System.Collections;
ArrayList lista; SortedList slista;
Queue coada; Stack stiva;
```

Cele mai importante metode deţinute de o colecţie sunt **`Add, Insert, Remove;`** dintre proprietăţi amintim **`Count`**.
Programul de mai jos exemplifică asemănările şi deosebirile dintre vectori şi colecţii. De remarcat cum o colecţie poate fi populată pornind de la un vector folosind metoda **`AddRange(),`** sau element cu element folosind metoda **`Add()`**. Preluarea din colecţie presupune şi un cast `((Pers)(lista[1])).Nume="Nou";` pe când din vector nu.

Şi vectorii şi colecţiile pot fi parcurse atât cu **`foreach,`** cât şi cu **`for,`** dimensiunea vectorului fiind dată de proprietatea **`Length,`** iar a colecţiei de proprietatea **`Count.`**

```
class Principal
{
 static void Main()
 {
     ArrayList lista; SortedList slista;
     Queue coada; Stack stiva;

     Pers [] vect = new Pers[3]
         {new Pers(0,"Zero"), new Pers(1,"Unu"),
                            new Pers(2,"Doi")};
     lista = new ArrayList();
     lista.AddRange(vect);
     // lista.Add(new Pers(0,"Zero"));
     // lista.Add(new Pers(1,"Unu"));
     // lista.Add(new Pers(2,"Doi"));

     lista[1]=lista[2]; vect[1]=vect[2];
     ((Pers)(lista[1])).Nume="Nou"; vect[1].Nume="Nou";
```



```
     foreach(Pers crt in lista)
         Console.Write("{0} - {1} \t",
                          crt.Marca, crt.Nume);
     Console.Write("\n");
     foreach(Pers crt in vect)
         Console.Write("{0} - {1} \t",
                          crt.Marca, crt.Nume);
     Console.Read();
 }
}
```

3. Proprietăţi

Proprietăţile combină un câmp cu metodele lui de acces; se puteau scrie şi funcţii obişnuite de acces, dar proprietăţile separă mai strict accesul în citire, de cel în scriere, simplificând adresarea câmpurilor private.

Sintactic, o proprietate se comportă ca o variabilă, dar în spate ascunde apeluri de accesori **get, set**

La folosire, proprietatea apare ca nume alternativ pentru un câmp; nu ocupă spaţiu separat, în memorie.

set primeşte implicit un argument ***value***, care este cuvânt cheie şi conţine valoarea folosită pentru modificarea câmpului privat.

O proprietate **este totdeauna publică** şi returnează valori, ca şi o metodă; ea permite accesul la un câmp şi chiar validări asupra datelor conţinute de câmp.

O clasă poate conţine numai unul sau ambii accesori pentru un câmp, asigurând acces în *read*, *write*, sau *read / write*.

Restricţii:

- nu pot fi declarate proprietăţi de tip `void`;
- fiind o încrucişare între câmp şi metodele sale de acces, proprietăţile nu au referinţă şi nu se pot transmite cu *ref* şi *out*, ca parametri în funcţii;
- nu se supraîncarcă, oferind câte o singură alternativă de acces pentru citire / scriere.

Exerciţiu

Să se exemplifice folosirea proprietăţilor pentru o clasă *Produs*.

Rezolvare
File / New / Project şi se alege tipul aplicaţiei **Console Application**
Numele aplicatiei: **My_App**

Se introduce textul de mai jos.

```
using System;
namespace proprietati
{
      class Produs
      {
            string den, um;
            float pret;
            int stoc;
            public Produs() {stoc=0;}

            public int p_stoc
            {
                  get { return stoc; }
                  set { if(value >=0) stoc = value; }
            }
      };

class Test
      {
        static void Main(string[] args)
            {
             Produs p1= new Produs();  p1.p_stoc=100;
             Console.WriteLine("\n Stocul este "+p1.p_stoc);
            }
      }
}
```

Variabila `stoc` fiind implicit privată, necesită proprietăți de acces; la stabilirea valorii ei se va avea în vedere că poate conține doar valori nenegative.

Observații.

- În C#, instanțele de clasă sunt referințe și trebuie asignate la obiecte generate cu **new**: `Produs p1 = new Produs();`
- listarea cu `Console.WriteLine` a variabilelor se poate face și concatenând un **string** cu o variabilă, fiind forțată astfel conversia la `string`: `Console.WriteLine("\n" + x );`

Proprietăți statice

Proprietățile pot fi statice, dacă ele sunt asociate unor câmpuri statice.

Exemplu

În clasa `Pers` există un câmp privat static **vmin**, care stă la baza validării vârstei unei persoane, validare făcută de asemenea printr-o funcție statică; **vmin** fiind static, are asociată tot o **proprietatea statică, VMIN**, la scriere.

```
namespace prop
{
 class Pers
  {
        private int varsta;
        static private int vmin;

        private static int valid(int v)
         {
          if ( v < vmin || v > 100)
              throw new ArgumentOutOfRangeException("varsta");
           return v;
          }

        public static int VMIN
                    { set { vmin = value; } }
        public Pers(int v ){ varsta = valid(v);}
  }

class Principal
      {
            static void Main(string[] args)
            {
                  Pers.VMIN = 16;
                  Pers p1;    p1 = new Pers(15);
                  Console.Read();
            }
      }
}
```

După cum se observă, calificarea se face pornind de la clasă, nu de la obiect : **Pers.VMIN = 16**; inițializarea s-a făcut în `Main()`, înainte de instanțierea vreunei persoane. Dacă în apelul constructorului s-ar da o vârstă mai mare ca 16, s-ar lansa automat o excepție de tip **OutOfRangeException**.

4. Delegări şi evenimente

Mecanismul delegării

delegate este tipul asociat unui obiect care încapsulează o referinţă la o metodă şi suplineşte conceptul de pointer de funcţie din C++ ; are forma generală: **delegate tip_rezultat nume(lista_param);**

```
delegate void PersDelegat(Produs p,double suma );
```

anunţă existenţa unui delegat numit `PersDelegat`, ce va stoca **referinţe de funcţii ce primesc obiecte de tip `Produs` şi o valoare de tip `double`.**

Instanţiat, ocupă spaţiu în memorie, capabil să stocheze referinţa uneia sau mai multor funcţii, nu rămâne doar o descriere formală (prototip):

```
PersDelegat Popescu = new PersDelegat
                              (a.cumpara_en_detail);
```

Prin iniţializare, delegatul a fost încarcat cu o referinţă (de funcţie) şi permite ulterior apelul metodei prin referinţă. Ulterior, delegatului i se pot ataşa şi alte metode, sau i se pot detaşa unele din metodele deţinute.

Metoda cu a cărei referinţă se încarcă delegatul poate fi **statică (de clasă)** sau **nestatică (de instanţă)**, conform signaturii, caz în care în nume apare o clasă (`Produs.cumpara`, adică delegare de a cumpăra orice produs), respectiv un obiect (`a.cumpara_en_detail`, adică delegare de a cumpăra doar *en detail* şi doar produse de tip `a`).

Referinţa poate fi modificată dinamic, delegatul permiţând apelul mai multor implementări ale unor metode care au acelaşi prototip.

De remarcat că deşi în prototipul delegării nu apare clasa care furnizează metoda, când se încarcă delegatul, adresa metodei trebuie calificată pornind de la obiect sau de la clasă, în funcţie de tipul ei. Dacă funcţia care invocă metoda (ex. `Main()`) face parte tot din clasa care furnizează metoda, metoda nu necesită calificare pornind de la clasă (sau obiect), deoarece ambele funcţii fac parte din aceeaşi clasă (sau acelaşi obiect, în cazul metodelor nestatice).

La încărcare se declară şi un nume de instanţă delegat (`Popescu`) moment în care metoda referită se dă ca parametru în constructor. Nu trebuie dat decât numele metodei, căci prototipul coincide cu cel al delegării, altfel ar fi semnalată eroare de sintaxă.

Delegatul permite **legarea întârziată**, adică amânarea precizării funcţiei de apelat, până la momentul execuţiei, când după ce se va încărca referinţa cu adresa uneia sau mai multor metode, se pot invoca metodele referite prin delegat:

```
Popescu(prod1,1000.0);
```

În plus, delegatul poate "călători" fiind transferat ca parametru în / din funcţii.

Exerciţiu

Să se definească şi să se utilizeze un delegat care să permită gestionarea metodelor de vânzare şi cumpărare, en gros şi en detail, în cadrul unor activităţi comerciale desfăşurate de o firmă.

```
using System;
namespace delegari
{
 delegate void PersDelegat(Produs p,double suma );
 class Produs
 {
     string den, um;
     public double pret;
     public int stoc;
     public Produs(){ stoc=0; pret=1.0;}
 }

 class Activ_comerciale
 {
  public void cumpara_en_detail(Produs p, double suma)
  {
  p.stoc+=(int)(suma / p.pret);
  Console.WriteLine("\nCumpara en detail de "+ suma+" lei");

  }

  public void cumpara_en_gros(Produs p,double suma)
   {
   p.stoc+=(int)(suma / (p.pret*0.9) );
   Console.WriteLine("\nCumpara en gros de "+ suma+" lei");
  }

  public void vinde(Produs p,double suma)
   {
     p.stoc-=(int)(suma / p.pret);
     Console.WriteLine("\n Vinde in valoare de "+suma);
   }
}
```

```
class Test
{
 static void Main(string[] args)
  {
     Produs prod1= new Produs();
     Activ_comerciale a=new Activ_comerciale();
     PersDelegat Popescu= new PersDelegat
                              (a.cumpara_en_detail);
     Popescu(prod1,1000.0);
     Console.WriteLine("\n Stocul este "+prod1.stoc+" buc");
     Popescu = new PersDelegat(a.cumpara_en_gros);
     Popescu(prod1,1000.0);
     Console.WriteLine("\n Stocul este "+prod1.stoc+
                                           " buc\n\n");
  }
 }
}
```

După cum se observă, declararea unui delegat se poate face în afara oricărei clase, deoarece înseamnă declararea unui tip; similar am fi putut declara o structură sau o clasă. Instanțierea delegatului se face însă doar în interiorul unei clase, deoarece în C# nu mai există variabile globale.

Delegatul ar fi putut fi declarat ca făcând parte dintr-o clasă, introducând astfel un tip interior unei clase, similar declarării unei clase într-o altă clasă. Numele delegatului va conține atunci și numele clasei. Spre exemplu, putem face declarația:

```
class Produs
 { // ....
    public delegate  void PersDelProd(Produs p,double suma );
   // ....
 }
```

iar în `Main()`, care face parte din altă clasă, instanțierea acestui tip se va face sub forma:

```
Produs.PersDelProd x = new Produs.PersDelProd
                             (a.cumpara_en_detail);
```

de unde se vede că tipul **`delegate`** se numește **`Produs.PersDelProd`**; aceasta deoarece în C# locul operatorului "::" este luat de operatorul "."

În ambele situații, apelul delegatului s-a făcut din interiorul clasei `Test`, astfel încât metoda folosită pentru inițializarea lui a necesitat calificarea cu obiectul **a** de tip `Produs`, iar invocarea nu a mai necesitat un calificator în fața numelui `Ionescu`. Dacă îl apelam din afara clasei `Test`, trebuia dat

numele clasei `Test`, dacă delegatul era static, sau numele unui obiect de tip `Test`, dacă delegatul era unul de instanță.

Multicasting este situația când pe un delegat aplicăm operatori += și -= pentru a adăuga și alte delegări (metode), respectiv pentru a elimina delegări.

Pentru ilustrarea facilității de multicasting în programul de mai sus declarăm un alt delegat și-l mandatăm să opereze toate genurile de activități comerciale:

```
PersDelegat Ionescu;
            Ionescu  = new PersDelegat(a.cumpara_en_gros);
            Ionescu += new PersDelegat(a.cumpara_en_detail);
            Ionescu += new PersDelegat(a.vinde);
            Ionescu(prod1,1000.0);
Console.WriteLine("\n Stocul este "+prod1.stoc+" buc");
```

Se observă un dezavantaj; la apelul **`Ionescu(prod1,1000.0);`** se execută tot setul de operații, dar pe aceleași date de intrare; ideal ar fi să controlăm fiecare apel individual de funcție din vectorul de funcții, astfel încât să putem particulariza după « i » atât funcția, cât și datele, dintr-o matrice de observații.

Evenimente

O clasă eveniment sau "publisher" este o clasă ce înglobează :

a. o instanță de **delegate**, căreia i se pune în față **`public event`**
b. o **funcție gen "fire"**, care activează delegatul, adică apelează o listă de funcții

Pentru test este nevoie:

- să definim efectiv un `delegate` numit și **`Handler`**, referință către o funcție de tip `void`, fără parametri:
  ```
  delegate void Handler();
  ```
- să declarăm o clasă "**publisher**" conținând evenimentul pe care îl va notifica public celor care subscriu (se anunță interesați de producerea lui) și o funcție de activare a metodelor comunicate de aceștia:
  ```
  class cls
  {
        public event Handler lst_metode;
        fire( );
  };
  ```

- să definim mai multe clase "**subscriber**", conținând o funcție cu care subscriu la eveniment (eventual și una cu care se detașează de la eveniment) și o metodă de prototipul delegatului, care doresc să le fie lansată automat în execuție, la producerea evenimentului:

```
class x
{
        public void Subscribe()        { /*... */ };
        public void Unsubscribe()      { /*... */ };
        void xH ()                     { /*... */ };
};

class y
{
        public void Subscribe()        { /*... */ };
        public void Unsubscribe()      { /*... */ };
        void yH ()                     { /*... */ };
};
```

Nu sunt restricții privind succesiunea derulării mecanismului de producere și tratare diferențiată a unui eveniment; uzual acest mecanism se desfășoară astfel:

1. clasă ***publisher*** anunță celor interesați, un **serviciu de notificare a unui eveniment**;
2. mai multe clase ***subscribers*** se abonează la serviciul oferit, comunicând funcțiile de tratare specifice fiecăreia;
3. la producerea evenimentului, de obicei depinzând de condiții exterioare claselor implicate, clasa ***publisher*** sesizează evenimentul, îl notifică abonaților activând lista de funcții comunicate de aceștia.

Pentru adaptare la context, de cele mai multe ori, instanța delegat nu se încarcă din timp, ci abia când clasa *subscriber* primește notificarea producerii evenimentului; astfel abonații își pot adapta în timp metodele de tratare a evenimentului, în funcție de context.

Observații:

- instanțiind clasă *publisher*, instanțiem și membrul delegat;
- instanțiind delegatul putem deja să începem stocarea de referințe de metode delegat;
- putem avea **mai multe clase** care subscriu la un eveniment;
- putem avea **mai multe instanțe ale aceleeași clase** care subscriu la un eveniment;

- handlere-le trebuie să fie cât mai generale, de obicei sunt fără argumente de apel și returnează *void*;
- dacă tipul `delegate` are parametri de intrare, **trebuie să îi și denumim**, de la început;
- evenimentul se apelează ca și delegatul, folosind ( ).

Fără folosirea **`delegate`**,(pointer de funcții) evenimentul ar fi fost "legat" indisolubil de o clasă și de numele unei metode, pe care trebuie să o lanseze în execuție; cu *delegate*, mai multe clase pot subscrie la același eveniment și pot folosi metode denumite diferit, pentru că apelul lor se va face prin pointer (referință).

Dacă în locul mecanismului de delegare s-ar fi folosit **`interface`** pentru funcțiile de apelat, urmând ca fiecare clasă interesată să își definească concret metodele ei, atunci am fi avut alte dezavantaje:

- metodele ar fi trebuit să fie totdeauna publice, adică expuse tuturor celor din exterior;
- n-am fi putut atașa / detașa metode, deoarece toate metodele unei interfețe fac parte obligatoriu din interfață .

Exercițiu

Pentru exemplificare, vom considera că recepția unui hotel deschide un serviciu automat, cel de "deșteptare" la o oră exactă. Clasa "publisher" **`Serviciu`**, care modelează acest lucru, conține o listă vidă ce va colecta ulterior referințele funcțiilor comunicate de cei care subscriu la acest serviciu, precum și o funcție gen "fire", prin care activează la diferite momente de timp, lista de funcții.

Prin constructor, clasa **`Serviciu`** publică oferta sa potențialilor interesați, adică două clase de data aceasta de tip "subscriber", **`Turist`** și **`Software`**, cărora le trimite adresa sa. Dispunând de această adresă, cele două clase se pot "abona", comunicând funcțiile ce doresc a fi lansate în execuție la producerea evenimentului: turistul este trezit fizic, iar software-ul este lansat în execuție, pe bază de timp.

Instanța **`delegate`** necesită în acest caz și parametri: ora și minutul momentului la care abonații doresc trezirea.

În clasa `Principal`, sunt instanțiati un turist **t1** și un program **prog1**; referințele lor, sub forma unui vector de obiecte generice (a se observa aici utilitatea derivării implicite a oricărei clase din clasa `object` și conversiile specifice) ajung la instanța clasei *publisher*, prin constructorul acesteia.

Când deșteptarea apare ca serviciu, clasa știe potențialii interesați (nu dezvolt noi servicii până nu identific potențialii beneficiari) și îi anunță, punându-și referința în "memoria" acestora. După caz, cei interesați subscriu sau nu la diverse momente de timp, sau în timp se detașează de la eveniment. Indiferent de toate acestea, într-o buclă *for* se simulează scurgerea timpului; serviciul recepție urmărește ora exactă și o notifică, dacă este momentul oportun, abonaților săi.

```
using System;
namespace test_evenim
{
      struct moment { public int o, m;}
      delegate void Trezeste(int h, int m);
      class Serviciu
      {
            public event Trezeste lista;

            public void Ora_exacta(int o,int m) // fire()
            {
                  if(lista!=null)          lista(o,m);
            }
            public moment [] vect_mom;
            public Serviciu( object [] vo)
            {
                  vect_mom = new moment[10];
                  ((Turist)vo[0]).s=this;     // se anunta
                  ((Software)vo[1]).s=this;   // clientilor
            }
      }

      class Software
      {
            public Serviciu s;

            public void Rulare(int h, int m )
            {
              if(h==s.vect_mom[1].o && m==s.vect_mom[1].m)
                 Console.WriteLine
                      ("\n Prog1 running start at "+h+":"+m);
            }
            public void Subscribe(int o, int m)
            {
                  s.lista+= new Trezeste(this.Rulare);
                  s.vect_mom[1].o= o;   s.vect_mom[1].m= m;
            }
      }
```



```
    class Turist
    {
          public Serviciu s;
          public void Desteptare(int h, int m )
          {
           if(h==s.vect_mom[0].o && m==s.vect_mom[0].m)
               Console.WriteLine
                   ("\n Turist t1 trezit la ora "+h+":"+m);
          }
          public void Subscribe(int o, int m)
          {
                s.lista += new Trezeste(this.Desteptare);
                s.vect_mom[0].o= o;  s.vect_mom[0].m= m;
          }
          public void Unsubscribe()
                {s.lista-= new Trezeste(this.Desteptare); }
    }
class Principal
{
  static void Main()
  {
    Software prog1=new Software(); Turist t1=new Turist();
    object []vo={t1,prog1}; // lumea obiectelor din jur
    Serviciu s=new Serviciu(vo);
    t1.Subscribe(11,30); prog1.Subscribe(0,30);
          //t1.Unsubscribe(); // are și detach()
    for(int h=0; h<24; h++)
          for(int m=0; m<60; m+=30)
           { // din 30 in 30 min notifică ora exactă
                  s.Ora_exacta(h,m);
           }
    Console.Read();
    }
  }
}
```

Evident, acesta nu este singura modalitate de expunere a derulării mecanismului evenimentelor, dar este una dintre implementările cele mai uzuale.

5. Interfețe

Interfețele permit separarea structurii unui obiect de modul în care acesta este folosit; adică separă implementarea unui obiect de funcționalitatea lui, tot așa cum la utilizarea unui autoturism trebuie să cunoaștem funcțiile acestuia și nu detaliile tehnice de realizare.

Ne amintim că existau cel puțin două motive care justificau moștenirea multiplă în C++ :

- primul era legat de faptul că o clasă derivată era în acelaşi timp de două sau de mai multe tipuri de bază (reamintim exemplul clasei `Pegas`, care era în acelaşi timp `Pasare` şi `Cal`);
- cel de-al doilea motiv era legat de generalitatea unor moşteniri de funcţii abstracte, cum ar fi serializarea obiectului, afişarea lui etc.

C# exclude moştenirea multiplă (derivarea din mai multe clase, simultan), dar permite totuşi **derivarea dintr-o clasă şi din mai multe interfeţe**, adică păstrează posibilitatea abstractizării pure a unor metode (cea de-a doua raţiune a moştenirii multiple), standardizând apelul acestora şi obligând în acest fel clasele să dea implementări publice pentru ele.

Interfeţele sunt un fel de clase, care conţin doar prototipuri de funcţii, numai că se declară cu **`interface`** în loc de `class`. Convenţional, numele lor începe cu "**I**".

Exerciţiu

Să se definească o interfaţă care să surprindă unitar principalele metode de deplasare, caracteristice deferitelor obiecte în mişcare.

Rezolvare

```
interface IMiscare
{
 void start();                   void stop();
 void inainte(int distanta,int viteza);
 void inapoi  (int distanta,int viteza);
 bool stanga(int unghi);         bool dreapta(int unghi);
}

class vehicul: IMiscare
 {
      string tip;
      public void start()
              { Console.WriteLine("\nStart vehicul"); }

      public void stop()
              { Console.WriteLine("\nStop vehicul"); }

      public void inainte(int distanta, int viteza)
        {
         Console.WriteLine
            ("\nVehicul {0} m inainte cu {1} km/h",
                                          distanta,viteza);
        }
```



```
      public void inapoi(int distanta, int viteza )
        {
         Console.WriteLine
            ("\nVehicul {0} m inapoi cu {1} km/h",
                                          distanta,viteza);
        }

      public bool stanga(int unghi)
        {
            Console.WriteLine
                  ("\nVehicul stanga cu {0} grade",unghi);
            return true;
        }

      public bool dreapta(int unghi)
        {
            Console.WriteLine
                  ("\nVehicul dreapta cu {0} grade",unghi);
            return true;
        }

    }

class Test
 {
   static void Main(string[] args)
      {
           vehicul v=new vehicul();
           v.start(); v.stanga(20);
           v.inainte(10,100);  v.stop();
           Console.Read();
      }
  }
```

O altă clasă, **`Pers`** spre exemplu, ar putea fi de asemenea derivată din **`IMiscare`**, implementând de manieră proprie aceeaşi interfaţă de mişcare.

La moştenirea dintr-o interfaţă, clasa trebuie să dea obligatoriu implementări pentru toate metodele interfeţei, cu respectarea strictă a numelui, tipului de retur şi parametrilor de apel, inclusiv a eventualilor specificatori **`out, ref`** etc.

Interfeţele:

- nu pot fi instanţiate, deoarece conţin metode doar cu prototipul lor, nu şi implementările lor;

- nu conţin câmpuri, nici chiar statice, ci doar **metode** şi eventual **proprietăţi** (care sunt combinaţie între câmp şi metode de acces), indexări şi evenimente (care sunt tot metode, dar mai speciale);
- nu dispun de constructori şi destructor, nefiind niciodată instanţiate;
- metodele nu poartă modificatori de acces, fiind totdeauna implicit publice;
- nu pot îngloba *struct*, *class* sau *enum*, căci acestea s-ar asocia unor câmpuri, iar câmpurile nu sunt admise într-o interfaţă;
- nu pot fi derivate dintr-o clasă sau structură (căci ar exista pericolul moştenirii unor date, lucru neadmis într-o interfaţă), ci numai eventual dintr-o altă interfaţă.

Dacă se doreşte ca o metodă moştenită dintr-o interfaţă să fie virtualizată la derivarea pe mai multe niveluri, prima **implementare efectivă** a acesteia într-o clasă, trebuie să menţioneze *virtual.*

O interfaţă se aseamănă cu o clasă abstractă, în sensul că ambele nu pot fi instanţiate, dar se deosebesc prin faptul că o clasă abstractă poate avea şi date membre, nu numai funcţii.

Chiar şi abstractă fiind, când o clasă este derivată dintr-o interfaţă trebuie să dea implementări pentru toate metodele moştenite din interfaţă.

Concluzionând, putem spune că utilitatea interfeţelor constă în faptul că:

- **standardizează funcţionalitatea** unor obiecte înrudite, dând prototipuri pentru unele metode prin care se lucrează cu acestea;
- obligă toate clasele să definească **public** şi în mod **specific** lor, aceste metode;
- permit o **adresare unitară** a acestor metode, folosind referinţe la interfaţa de bază, din care a fost derivat obiectul.

C++	C#
Limbaj mixt procedural / obiectual: - se pot defini variabile şi constante de tipuri fundamentale, masive şi obiecte (tipul *struct* şi *class*); - programul poate conţine funcţii independente, clase şi structuri.	Limbaj pur obiectual: - toate variabilele cât şi constantele sunt tratate ca obiecte; - programul este format numai din clase şi structuri.
In funcţie de tipul de aplicaţie, punctul de intrare este diferit: - *main*() – pentru aplicaţii în mod consolă; - *WinMain*() – pentru aplicaţii în Windows în tehnologie SDK; - *MyApp x;* - definirea unui obiect de clasă derivată din clasa CApp pentru aplicaţii Windows folosind MFC.	Indiferent de tipologia aplicaţiei o clasa trebuie să definescă o metodă statică numită *Main*().
Se manipulează adrese, ca pointeri sau ca referinţe.	Se utilizează referinţe; în modul de lucru *unsafe* se pot utiliza şi pointeri.
Nu există implementat tipul şir de caractere ca tip fundamental; diverse biblioteci de clase implementează acest tip (ex. *string* în STL sau *CString* în MFC).	Tipul şir de caractere este introdus prin clasa *string*.
Metodele unei clase se pot defini în interiorul unei clase (funcţii *inline*), cât şi în exteriorul ei.	Metodele se definesc doar în interiorul clasei.
O clasă poate conţine doar variabile membru şi cod.	O clasă poate conţine doar variabile membru, proprietăţi şi cod.
Incapsularea datelor şi a codului se face doar în structuri şi clase.	La fel şi în C#, numai că există o diversitate de clase: *class*, *interface*, *abstract*.
Se permite moştenirea multiplă.	Nu se permite moştenirea multiplă la nivel de clasă concretă (*class*). Se pot moşteni o clasă şi mai multe interfaţe.

Tab. 1.1 Sinteza principalelor deosebiri dintre limbajele C# şi C++

IMPLEMENTAREA ÎN C# A STRUCTURILOR DE DATE DINAMICE

1. Aspecte generale
2. Implementarea structurilor de date dinamice liniare
3. Implementarea structurilor de date dinamice arborescente

1. Aspecte generale

In C#, ca şi în alte implementări ale limbajului C (C++), sunt definite clase de tip colecţie, pentru a facilita lucru cu structurile de date dinamice cum ar fi: lista, stiva, coada etc. Aceste preocupări au fost concretizate prin existenţa unor clase specializate, ca de exemplu, în MFC: `CArray` – pentru lucru cu masive, `CList` – pentru lucru cu liste, stive, cozi şi `CMap` – pentru implementarea structurii de `hash` (tabele de dispersie); alteori aceste definiri au făcut obiectul unor biblioteci specializate, ca de exemplu STL (Standard Template Library). In toate implementările s-a urmărit în mod deosebit asigurarea unei cât mai mari generalităţi (din punct de vedere al tipului informaţiei utile) şi a unei cât mai mari flexibilităţi (prin construirea de algoritmi generici).

În C++, generalitatea este asigurată prin faptul că se pot defini şabloane de clase (clase `template`), iar flexibilitatea se asigură prin scrierea unor secvenţe de cod, trimise ca parametri altor funcţii, prin intermediul pointerilor la funcţii.

C# fiind un limbaj pur obiectual, orice obiect care este de tip fundamental sau de tip utilizator (tip abstract de date) derivă din clasa **`Object`** adică poate fi manipulat şi ca tip **`object`**. Prin manipularea variabilelor de tip **`object`**, în limbajul C# se asigură o maximă generalitate colecţiilor de date. Fiind vorba de un limbaj pur obiectual, înseamnă că nu se pot utiliza funcţii independente, de aceea inducerea anumitor secvenţe de cod se face prin intermediul interfeţelor.

In C# următoarele clase de tip colecţie implementează principalele structuri de date dinamice: **`ArrayList`** – implementează structura de listă, **`Stack`** – implementează structura de stivă, **`Queue`** – implementează structura de coadă şi **`HashTable`** – implementează structura de `hash`;

Acest capitol nu îşi propune să prezinte implementările tuturor structurilor de date dinamice şi nici să definească toate operaţiile aferente

Implementarea în C# a structurilor de date dinamice

lor. Scopul lui este de a implementa structuri de date dinamice, prin prisma obiectelor de tip colecţie din C#.

2. Implementarea structurilor de date dinamice liniare

Toate clasele vor fi definite în domeniul de nume **`structuri_de_date_dinamice`**. Clasa **`lista`** se va defini ca o colecţie de obiecte **`nod`**. Un nod se va implementa prin intermediul clasei **`nod`** care va aparţine domeniului de nume **`noduri`**, ce va fi inclus în domeniul de nume **`structuri_de_date_dinamice`**.

Un nod conţine:

- **ca date**, informaţia utilă (**`inf`**) de tip `object` şi legătura (**`next`**) către nodul următor, de tip referinţă la `nod`;

```
namespace structuri_de_date_dinamice
{
 namespace noduri
 {
      public class nod
      {
            object inf;
            public nod next;
            public nod(object k)  {  inf=k;    next=null;  }
            public nod(object k, nod urm){inf=k; next=urm;}
            public object informatia
             {
                  get    {   return inf; }
                  set    {   inf=value;  }
             }
      }
            //...
 }
}
```

- **constructori**, unul pentru a iniţializa informaţia utilă din parametrul de apel şi legătura cu **`null`** şi altul care iniţializează atât informaţia utilă, cât şi legătura preluată prin parametru de apel;
- **proprietatea `informatia`** pentru a obţine / modifica informaţia utilă.

Structura de tip listă

Lista se va implementa prin intermediul clasei **`lista`** care aparţine direct domeniului de nume **`structuri_de_date_dinamice`** şi care are ca membri:

- **capul listei** (`cap`) ce este o referință către primul element al listei și **numărul de noduri** din listă (`nrn`);
- **constructorul** care crează lista vidă;
- metodele: `Adaugare()` – pentru a adăuga un element în listă și `Afisare()` – pentru afișarea elementelor listei;
- proprietatea `NrNoduri` obține numărul de noduri din listă și se poate folosi și pentru test de listă vidă.

```
namespace structuri_de_date_dinamice
{
 namespace noduri
  {
      public class nod  {     // ... definita anterior      }
  }
 public class lista
  {
      protected noduri.nod cap;
      protected uint nrn;

      public lista() { cap=null; nrn=0; }

      public void Adaugare(object k)
            {
                  if(cap==null) cap=new noduri.nod(k);
                  else
                  {
                        noduri.nod aux=cap;
                        while(aux.next!=null)   aux=aux.next;
                        aux.next=new noduri.nod(k);
                  }
                  nrn++;
            }

     public void Afisare()
      {
        noduri.nod aux=cap;
        while(aux!=null)
         {
          Console.Write("  {0}  ",aux.informatia);
          aux=aux.next;
         }
      }

     public uint NrNoduri   {   get     {  return nrn;  } }
 }
}
```


Utilizarea acestei clase se poate exemplifica prin secvența următoare:

```
class Principala
{
 static void Main(string[] args)
 {
  structuri_de_date_dinamice.lista l=
             new structuri_de_date_dinamice.lista();
  l.Adaugare(20); l.Adaugare(90); l.Adaugare(45);
  l.Afisare();
 }
}
```

Cei care au folosit obiecte de tip colecție în C# au văzut că se foloseau metode standard pentru operațiile uzuale. Astfel, pentru adăugarea unui element în colecție se utilizează metoda `Add()`, pentru ștergere `Remove()`, pentru căutare `Contains()` etc. Alinierea la acest standard se face prin definirea clasei `lista` ca fiind derivată din interfața `IList`:

```
public class lista : IList   { ...   }
```

Stergerea primei apariții a unui element din listă se efectuează de către metoda `sterge` care primește ca parametru informația utilă a nodului de șters și returnează un boolean cu semnificația `true`, dacă s-a șters un nod și `false`, altfel. Această metodă apelează metoda privată `sterg` care efectuează ștergerea propriu-zisă a nodului, în manieră recursivă.

```
public class lista
{
// ...
  bool sterg(ref noduri.nod str, object o)
  {
      if(str == null) return false;
       else
            if(str.informatia.Equals(o))
             {
                 nrn--;   str=str.next;   GC.Collect();
                 return true;
             }
            else   return sterg(ref str.next,o);
  }

 public bool Sterge(object o)
              {    return sterg(ref cap,o); }
}
```

S-a precizat că se şterge un nod dacă informaţia lui utilă coincide cu valoarea parametrului de apel. Informaţiile utile fiind de tip **object** trebuie definită egalitatea dintre două obiecte de tip object. Acest lucru se realizează prin supradefinirea metodei **Equals()**, care aparţine clasei **Object** ce este clasă de bază pentru orice altă clasă din C# şi introduce tipul object. Pentru informaţii utile de tipuri fundamentale (int, string, uint, double etc) metoda **Equals()** este definită, astfel încât nu sunt necesare alte redefiniri. In cazul în care dorim ca informaţia utilă să fie de un tip introdus prin intermediul unei clase de utilizator, atunci această metodă trebuie supraîncărcată corespunzător.

Ca exemplu, vom defini clasa **persoana** cu membrii:
- numele (**num**) şi vârsta (**v**) persoanei;
- constructorul care iniţializează datele din parametrii de apel;
- supradefinirea metodei **ToString()** pentru a converti o persoana la un şir de caractere, utilă la afişarea unui obiect persoana;
- supradefinirea metodei **Equals()** pentru a testa dacă două persoane sunt egale; se consideră că două persoane sunt egale dacă au aceeaşi vârstă;
- proprietatea **Nume** pentru a accesa numele persoanei.

```
public class persoana
{
        string nume;
        int v;
        public persoana(string np, int t){  nume=np; v=t;      }

        public override string ToString()
               {    return " "+nume+" "+v.ToString();    }

        public override bool Equals(object p)
               {     return v==((persoana)p).v;          }

        public string Nume  {    get  {    return nume;    }     }
}
```

Considerând aceste aspecte, clasa lista este funcţională atât pentru informaţii utile de tipuri fundamentale, cât şi pentru tipuri abstracte de date, după cum se poate observa în secvenţele următoare:

```
//testare lista pe date elementare
        structuri_de_date_dinamice.lista l =
              new structuri_de_date_dinamice.lista();
        l.Adaugare(20); l.Adaugare(90); l.Adaugare(45);
        l.Afisare();
```



```
        if(l.Sterge(90)) Console.WriteLine(" Nod sters!! ");
         else Console.WriteLine(" Nod inexistent!! ");
        l.Afisare();

//testare lista pe tipul persoana
        structuri_de_date_dinamice.lista lp =
              new structuri_de_date_dinamice.lista();
        lp.Adaugare(new persoana("Ionescu", 33));
        lp.Adaugare(new persoana("Gigi", 12));
        lp.Afisare();
        if(lp.Sterge(new persoana(" ",12)))
               Console.WriteLine(" Nod sters!! ");
         else  Console.WriteLine(" Nod inexistent!! ");
        lp.Afisare();
```

In C# elementele unei colecţii pot fi accesate în vederea prelucrării lor folosind instrucţiunea **foreach**. Pentru colecţia noastră, definită prin intermediul clasei lista, instrucţiunea foreach nu funcţionează. Acest lucru devine posibil făcând următoarele completări:
- clasa **lista** va moşteni interfaţa **IEnumerable**;
- ca o consecinţă a punctului anterior este necesar să se definească metoda **GetEnumerator()**, în clasa lista, după cum urmează:

```
public IEnumerator GetEnumerator()
     { return nrn==0 ? null  : new ListEnum(cap); }
```

In cazul în care lista e vidă, metoda returnează null, altfel metoda întoarce un obiect care implementează interfaţa **IEnumerator** pentru o listă. Obiectul este de clasă **ListEnum** şi referă elementele listei prin intermediul capului listei (cap);

- se defineşte clasa **ListEnum** care implementează interfaţa IEnumerator în domeniul de nume **structuri_de_date_dinamice**:

```
class ListEnum  : IEnumerator
 {
   noduri.nod aux, init;
   bool vb;

   public ListEnum(noduri.nod ccp)
        { aux=init=ccp; vb=true; }

   public object Current {get {return aux.informatia;} }
```

```
    public bool MoveNext()
          {
                if(aux==init && vb )
                     { vb=false;  return true;        }
                if (aux.next!=null)
                  {   aux=aux.next;     return true; }
                else return false;
          }

    public void Reset() { aux=init;  vb=true;        }
  }
```

Din codul sursă se observă următoarele:

- constructorul inițializează membrii de tip **nod** cu capul listei și variabila booleană **vb** cu **true**;
- metoda **Reset()** repune referința auxiliară **aux** pe începutul listei și **vb** pe **true**;
- proprietatea **Current** furnizează în afară informația utilă a nodului curent (cel referit prin **aux**);
- metoda **MoveNext()** are ca scop referirea următorului nod valid al listei, caz în care returnează **true**; dacă nu mai există noduri în listă returnează **false**.

Instrucțiunea **foreach** începe prin a apela metoda **Reset()**, după care apelează metoda **MoveNext()** cu scopul de a referi primul element al colecției, adică primul nod valid al listei; scopul variabilei **vb** este de a face distincție între primul apel al metodei **MoveNext()** și celelalte, care vor avansa referința **aux** atâta timp cât mai există un nod valid în listă. Utilizarea instrucțiunii **foreach** împreună cu colecția **lista** se poate face acum sub forma:

```
    try
    {
      foreach(persoana pers in lp)
                Console.WriteLine(pers.ToString());
    }

    catch
     {
          Console.WriteLine("Lista vida!!!");
     }
```

S-a inclus instrucțiunea `foreach` într-un bloc `try` cu scopul de a lansa o excepție în cazul în care lista este vidă.

Structura de tip stivă

Clasa **stiva** este definită tot în domeniul de nume `structuri_de_date_dinamice` și este **derivată din clasa lista** și operează cu disciplina LIFO (ultimul intrat, primul ieșit).

```
public class stiva : lista
 {

  public void Push(object o)
   {
     noduri.nod temp = new noduri.nod(o,cap);
     nrn++; cap=temp;
   }

 public object Pop()
  {
     if(NrNoduri==0)
           throw new Exception(" !! Stiva Vida !! ");
     else
           {
                 object temp = cap.informatia;
                 cap=cap.next;   nrn--;   GC.Collect();
                 return temp;
           }
   }
 }
```

- metoda **Push**() inserează un nod în capul listei (vârful stivei); informația utilă, de tip `object` este dată ca parametru în metodă;
- metoda **Pop**() extrage elementul din vârful stivei și returnează informația lui utilă. In caz că stiva este vidă se aruncă o excepție cu mesajul "*Stiva vida*".

Pentru a testa dacă stiva este vidă se folosește proprietatea **NrNoduri** moștenită de la clasa `lista`.

O secvență de cod în care se utilizează clasa stiva ar putea arăta astfel:

```
structuri_de_date_dinamice.stiva s =
            new structuri_de_date_dinamice.stiva();
s.Push(34); s.Push(56); s.Push(49);

//extragere din stiva cu terminare prin exceptie
while(true)
{
      try { x=(int)s.Pop(); }
      catch(Exception er)
```

```
        {    Console.Write(er.Message);  break;   }
        Console.Write(" "+ x.ToString()+" ");
    }

    //extragere cu terminare prin test de stiva vida
    while(s.NrNoduri!=0)
      {
        x=(int)s.Pop();
        Console.Write(" "+x.ToString()+" ");
      }
```

Structura de **coadă** se poate implementa tot printr-o clasă derivată din clasa `lista`, numai că disciplina de lucru este FIFO (primul intrat primul ieşit) şi se descriu metodele corespunzătoare.

3. Implementarea structurilor de date dinamice arborescente

Ca şi în cazul listei, arborele poate fi descris tot ca o colecţie de noduri. În cazul unui **arbore binar**, un nod conţine informaţia utilă şi are cel mult doi descendenţi. Se poate descrie clasa ce implementează nodul arborelui binar (`noda`) ca fiind derivată din clasa `nod` a listei şi având în plus încă o legătură (`nextd`), pe care o considerăm către subarborele drept. Şi această clasă se va defini tot în domeniul de nume **`noduri`**.

```
namespace structuri_de_date_dinamice
{
  namespace noduri
  {
      public class nod  {      //definita anterior      }
      public class noda: nod
      {
           public noda nextd;
           public noda(object k): base(k)    { nextd=null;   }
           public bool este_frunza()
                 { return next==null && nextd==null;  }
      }
  }
  // ...
}
```

Observaţii.

- constructorul clasei **`noda`** iniţializează informaţia utilă din parametrul de apel şi pune legăturile pe `null`;
- metoda **`este_frunza()`** testează dacă nodul este sau nu frunză.

Un **arbore binar de căutare** este un arbore binar (fig. 2.1) în care se respectă următoarele reguli:

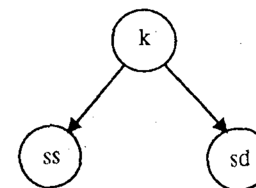


Fig. 2.1 Arbore binar

- fie arborele este vid;
- fie cheia `k` din nodul rădacină este mai mare decât toate cheile din subarborele stâng (`ss`) şi mai mică decât toate cheile din subarborele drept (`sd`); la rândul lor subarborii `ss` şi `sd` formează tot arbori binari de căutare.

Clasa care implementează structura arbore binar de căutare (**`arbbin`**) aparţine domeniului de nume **`structuri_de_date_dinamice`** şi conţine:

- **ca date**, rădacina arborelui (**`rad`**) şi numărul de noduri (**`nrn`**);
- **constructorul** care crează arborele vid;
- **proprietatea `NrNoduri`,** care furnizează numărul de noduri ale unui arbore; se poate folosi şi în scopul de a testa dacă un arbore este sau nu vid;
- **metodele**:
  - **`Adaugare()`** pentru a insera un nod în arbore; apelează metoda privată **`ins()`**, care efectuează inserarea propriu-zisă a nodului în arbore, în manieră recursivă;
  - **`Afisare()`** pentru a afişa arborele; apelează metoda privată **`afis()`**, care afişează propriu-zis arborele în forma `Rădăcină(Stînga,Dreapta)`, în manieră recursivă.

```
public class arbbin
 {
  noduri.noda rad;
  uint nrn;
  noduri.noda ins(noduri.noda r, object o, IComparer comp)
   {
     if(comp==null)comp=Comparer.Default;
     if(r==null)
     {
       nrn++;
```

```
        return new structuri_de_date_dinamice.noduri.noda(o);
      }
      else
      {
        if(comp.Compare(o,r.informatia)<0)
          r.next=ins((noduri.noda)r.next,o,comp);
        else
          if(comp.Compare(o,r.informatia)>0)
            r.nextd=ins(r.nextd,o,comp);
        return r;
      }
    }

  void afis(noduri.noda r)
    {
      if(r!=null)
       {
        Console.Write("    "+r.informatia.ToString()+"    ");
        if(!r.este_frunza())
         {
          Console.Write("(");    afis((noduri.noda)r.next);
          Console.Write(",");    afis(r.nextd);
          Console.Write(")");
         }
       }
      else Console.Write("-");
    }

  public arbbin() { rad=null; nrn=0; }

  public void Adaugare(object o, IComparer comp)
      {     rad=ins(rad,o,comp);   }

  public void Afisare() { afis(rad); }

  public uint NrNoduri   { get     { return nrn;   }      }
}
```

Inserarea unui nod în arbore trebuie să respecte disciplina pe care o impune un arbore binar de căutare, deci operația are la bază comparații între informațiile utile din nodurile arborelui și valoarea de inserat. Informațiile utile sunt de tip `object`, deci comparația nu se poate face direct, ci prin intermediul interfeței **`IComparer`**.

Se observă la metodele **`Adăugare()`** și **`ins()`** că ultimul parametru de apel este un obiect care implementează interfața **`IComparer`**. Prin intermediul acestui obiect se face comparația între două variabile de tip

`object`, mai precis prin apelul metodei **`Compare()`** care primește două obiecte de tip `object` și returnează un întreg cu semnificația:

- < 0 dacă primul obiect este mai mic;
- = 0 dacă cele două obiecte sunt egale ;
- \>0 dacă primul obiect este mai mare.

Dacă se lucrează cu obiecte de tip fundamental, atunci se folosește clasa `Comparer` care implementează interfața `ICompare` pentru aceste tipuri. In această situație parametrul de apel de tip `ICompare` va fi `null`, iar interfața se va extrage prin invocarea proprietății statice `Default` a clasei `Comparer`:

```
if(comp==null)  comp=Comparer.Default;
```

Următoarea secvență de cod prezintă modul în care se poate folosi clasa **`arbbin`** pentru lucru cu arbori binari de căutare, având informații utile de tip fundamental.

```
//informatie utila de tip fundamental
structuri_de_date_dinamice.arbbin a =
       new structuri_de_date_dinamice.arbbin();
a.Adaugare(15,null);  a.Adaugare(20,null);
a.Adaugare(3,null);   a.Adaugare(18,null);
a.Afisare();
```

Dacă se folosesc tipuri abstracte de date, induse prin intermediul claselor de utilizator, atunci trebuie definită interfața **`IComparer`** în mod corespunzător. De exemplu, pentru clasa **`persoana`**, definită anterior, considerăm că operația de comparare se face la nivelul numelui persoanei; în acest caz, numele fiind de tip elementar, se poate invoca metoda **`CompareTo()`** specifică acestui tip:

```
public class Pers_Comp : IComparer
{
  public int Compare(object x, object y)
  {
   return ((persoana)x).Nume.CompareTo(((persoana)y).Nume);
  }
}
```

Utilizarea arborelui binar cu informații utile de tip `persoana` având definită interfața `IComparer` prin clasa `Pers_Comp` se face sub forma:

```
persoana[] p =
```

```
    {
     new persoana("Ionescu", 13), new persoana("Adam", 33),
     new persoana("Vasile", 38)
    };

structuri_de_date_dinamice.arbbin b =
        new structuri_de_date_dinamice.arbbin();

Pers_Comp pc = new Pers_Comp();

b.Adaugare(p[0],pc);  b.Adaugare(p[1],pc);
b.Adaugare(p[2],pc]);

b.Afisare();
```

O altă modalitate de definire a unei interfețe pentru propria noastră clasă constă în a deriva clasa din interfața respectivă. Astfel, se definește clasa **persoana** ca fiind derivată din interfața **IComparer** și se implementează obligatoriu metoda:

```
            public int Compare(object , object )
```

Clasa **persoana** se descrie acum astfel:

```
public class persoana : IComparer
{
 // datele, metodele si proprietatile sunt similare
 // celor din definirea anterioara a clasei persoana

public int Compare(object x, object y)
  {
    return ((persoana)x).Nume.CompareTo(((persoana)y).Nume);
  }
}
```

Folosirea obiectului de tip persoana pe post de informație utilă în arbore se poate face ca în secvența:

```
      persoana[] p=
       {
            new persoana("Ionescu", 13),
            new persoana("Adam", 33),
            new persoana("Vasile", 38)
        };

      structuri_de_date_dinamice.arbbin b =
            new structuri_de_date_dinamice.arbbin();
```



```
      b.Adaugare(p[0],p[0]);  b.Adaugare(p[1],p[0]);
      b.Adaugare(p[2],p[0]);

      b.Afisare();
```

Concluzii

In C# lucru cu colecții, în această formă, a fost generalizat în sensul că s-a folosit pentru implementarea structurilor de date dinamice, gestionarea datelor din controalele bazate pe colecții (ListBox, ComboBox, ListView etc.) și a înregistrărilor dintr-un set de date (DataSet). Cunoașterea aspectelor necesare definirii obiectelor de tip colecție este utilă pentru programatorii de C#, deoarece ei pot să-și dezvolte acum propriile biblioteci de clase sau chiar biblioteci de controale, după modelul definit în mediul .NET.

Exercițiu

Să se completeze clasa **arbbin** astfel încât instrucțiunea **foreach** să se aplice și pe obiecte ale acestei clase și să echivaleze cu parcurgerea în inordine (stânga, rădăcină, dreapta) a arborelui binar de căutare.

INTRODUCERE ÎN WINDOWS FORMS

1. Crearea unei aplicații Windows
2. Derivarea controalelor
3. Modificarea dinamică a proprietăților
4. Utilizarea indexărilor

1. Crearea unei aplicații Windows

Crearea unei aplicații Windows se poate face fie scriind direct cod sursă, fie vizual, folosind mediul Visual C# care scrie automat o mare parte din codul sursă, în funcție de tipologia aplicației alese. Indiferent de maniera prin care se ajunge la codul sursă, el rămâne elementul de bază, fiind suficient pentru a reconstitui oricând sub mediul integrat, ultima formă a aplicației.

1.1 Crearea unei aplicații Windows prin scriere de cod sursă

Pentru a înțelege mai bine ce instrucțiuni sunt scrise automat, când din mediul vizual dăm diverse comenzi, dragăm controale pe suprafața formei sau atașăm funcții de tratare a unor evenimente, vom încerca să facem noi înșine acest lucru, scriind minimul de cod sursă necesar.

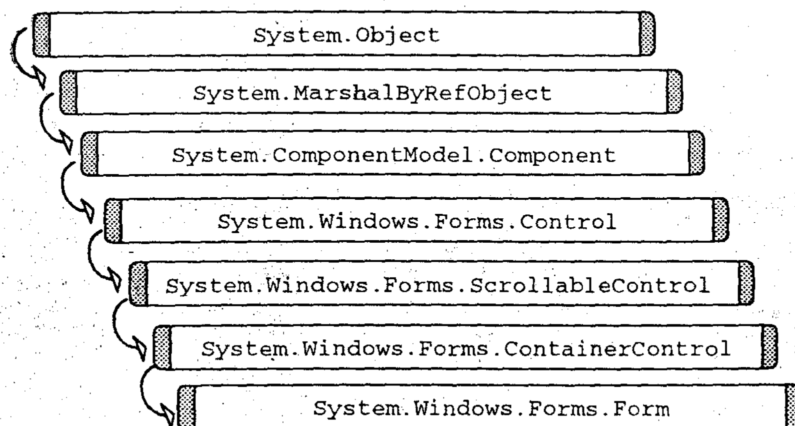


Fig. 3.1 Ierarhia de derivare a clasei Form

Cheia înțelegerii aplicațiilor de tip Windows Forms o reprezintă clasa `System.Windows.Forms.Form`; în structura de derivare specifică bibliotecii de clase din .NET, ea este derivată printre altele din `System.ComponentModel.Component` și deci fiind o componentă poate interacționa cu alte componente, în maniera controalelor; fiind o clasă derivată și din `System.Windows.Forms.ContainerControl` ea poate ține alte controale.

Din denumirile de mai sus, doar ultimul cuvânt este nume de clasă, celelalte sunt denumiri de namespace-uri, adică subdomeniul care introduce definirea clasei și domeniile din care face el parte. Acest tip de calificare asigură recunoașterea clasei chiar dacă nu anunțăm în preambulul programului, prin `using`, domeniile de nume utilizate.

Este suficient pentru a avea o fereastră Windows, să derivăm propria noastră formă din `System.Windows.Forms.Form` și s-o instanțiem în momentul lansării în execuție a aplicației:

```
using System;
using System.Windows.Forms;

public class Form1: Form
{
  public Form1()
     {
           // Size = new System.Drawing.Size(500,300);
           // Text = "Forma simpla";
     }
  public static void Main()

     {
           Application.Run(new Form1());
     }
}
```

Putem să nu punem nimic în constructor (liniile sursă comentate) și programul tot afișează forma, dar cu coordonate și dimensiuni implicite. Compilarea și linkeditarea unui astfel de program se poate face din linie de comandă:

```
csc /target:winexe /r:System.dll /r:System.Drawing.dll
                   /r:System.Windows.Forms.dll Forma1.cs
```

sau direct din mediul integrat ; în acest scop este suficient ca într-o aplicație C# Windows Application să substituim codul din fisierul `xxx.cs` cu codul de mai sus și să cerem din meniu **Debug / Start without debugging**.

Dacă redimensionăm forma în *Designer*, mediul adaugă imediat o funcție numită `InitializeComponent()`, care notează noile coordonate ale formei; tot mediul pune și apelul funcției în constructorul formei, astfel încât forma să beneficieze la trasare de noile coordonate.

Vom modifica programul de mai sus pentru a scrie ceva peste forma afișată. Pentru aceasta, vom intercepta mesajul **WM_PAINT**, suprascriind funcția `OnPaint()`.

```
using System;
using System.Windows.Forms;
using System.Drawing;

public class Form1: Form
{
  public Form1()
      {
            Text = "Forma scrisa";
      }
  protected override void OnPaint(PaintEventArgs e)
      {
        e.Graphics.DrawString
          ("Scriere pe forma", Font,
                     new SolidBrush(Color.Black), 100, 100);
      }
  public static void Main()
      {
            Application.Run(new Form1());
      }
}
```

`OnPaint()` este o funcție moștenită de formă și suprascrisă aici, care tratează mesajul **WM_PAINT,** transmis de sistem ori de câte ori o fereastră trebuie retrasată. Forma fiind în fond tot o fereastră, va primi și ea mesajul atunci când suprafața ei trebuie retrasată (spre exemplu, o altă fereastră a suprapus o porțiune din formă sau s-a dat minimize / maximize).

`OnPaint()` primește ca parametru un obiect de tip `PaintEventArgs` ce ține argumentele necesare retrasării ferestrei ; vom prelua din acest obiect, referința la contextul grafic al ferestrei, folosit pentru afișare șir de caractere.

`Font` este o proprietate a formei (forma are un font implicit, care poate fi modificat ulterior); pensula și coordonatele unde se face scrierea se instanțiază direct, în apelul funcției.

Programul de mai sus folosește o legătură moștenită, între un eveniment **WM_PAINT** și o funcție de tratare `OnPaint()`. Vom încerca acum să adăugăm noi o legătură între un eveniment declanșat de utilizator prin apăsarea mouse-ului pe suprafața formei (`MouseDown`) și o funcție de tratare a lui. Avem nevoie de două lucruri:

- să scriem o funcție de tratare a acestui eveniment; spre exemplu, la `MouseDown` să se înregistreze noua poziție a mouse-ului și să cerem rescrierea textului la noua poziție, prin invalidarea dreptunghiului ocupat de fereastră pe ecran:

```
protected void Form1_MouseDown
                              (object sender,MouseEventArgs e)
   {
         x = e.X;                y = e.Y;
         Invalidate();
   }
```

- să înregistrăm (în funcția `InitializeComponent()`, sau într-o altă funcție) la nivelul formei, legătura dintre **evenimentul** `MouseDown` și **funcția de tratare** propusă de noi, atașându-i delegatului `MouseDown` și numele funcției noastre:

```
MouseDown += new MouseEventHandler (Form1_MouseDown);
```

Programul complet va arăta acum astfel:

```
using System;
using System.Windows.Forms;
using System.Drawing;

public class Form1: Form
{
      private void InitializeComponent()
      {
            MouseDown +=
                   new MouseEventHandler (Form1_MouseDown);
      }

      private float x, y;
      private Brush stdBrush;
      public Form1()
      {
            InitializeComponent();
            Size = new System.Drawing.Size(300,200);
            Text = "Forma cu rescriere";
            x = y = 10;
            stdBrush = new SolidBrush(Color.Black);
      }
```

```
protected void Form1_MouseDown
                    (object sender, MouseEventArgs e)
{
        x = e.X;            y = e.Y;
        Invalidate();
}

protected override void OnPaint(PaintEventArgs e)
{
        e.Graphics.DrawString("Scriere pe forma", Font,
                    new SolidBrush(Color.Black), x, y);
}

public static void Main()
{
        Application.Run(new Form1());
}
}
```

După cum se observă, acum apăsarea cu mouse-ul declanşează şi retrasarea formei, prin invalidarea dreptunghiului ferestrei ce conţine forma; retrasarea propriu-zisă se face prin invocarea metodei `OnPaint()`.

Gama evenimentelor este foarte variată, cuprinzând de la evenimente produse de utilizator (cum ar fi acţionări de taste sau mouse, stoparea execuţiei etc.) până la evenimente ce nu depind de utilizator (recepţionarea unui mail, modificarea unui fişier, terminarea altui program, scurgerea unei cuante de timp etc).

Sintetizând, putem concluziona că tratarea unui eveniment se poate face prin două modalităţi:

1. **ataşarea unei metode la delegatul specific evenimentului:**

```
MouseDown +=
          new MouseEventHandler (Form1_MouseDown);
```

2. **suprascrierea unei metode `protected` moştenită din clasa de bază:**

```
protected override void OnMouseDown
            ( System.Windows.Forms.MouseEventArgs e )
{
      base.OnMouseDown(e ) ;
      // cod sursa propriu funcţiei
}
```


Folosind mecanismul bazat pe suprascrierea metodei moştenite, nu mai funcţionează mecanismul bazat pe handler, aceasta deoarece metoda `OnMouseDown` originală era cea care activa şi metodele din delegat.

Pentru a funcţiona ambele mecanisme de tratare evenimente, se recomandă ca la suprascrierea unei metode să apelăm mai întâi varianta din bază şi abia apoi să scriem codul specific:

```
base.OnMouseDown(e ) ;
```

lucru pe care ni-l sugerează de cele mai multe ori şi sistemul, punând automat în funcţie, apelul de mai sus.

Aplicaţiile de până acum nu conţineau alte controale pe formă. Când acestea sunt preluate vizual din trusă cu instrumente (ToolBox) lucrurile sunt simple; ca să le creem prin program însă, va trebui să scriem cod sursă pentru:

- **declararea în clasa Form a unei referinţe**, prin care să gestionăm controlul respectiv:

```
private System.Windows.Forms.Button btn1;
```

- **instanţierea controlului,** în constructorul formei şi stabilirea principalelor proprietăţi dorite:

```
btn1 = new System.Windows.Forms.Button();
btn1.Location = new System.Drawing.Point(100, 200);
btn1.Text = "Creat soft";
```

- **Adăugarea controlului la colecţia de controale** a formei

```
Controls.Add(btn1);
```

Controlul este acum vizibil la momentul execuţiei aplicaţiei. Ataşarea unor metode de tratare evenimente se poate face după acelaşi model ca la formă:

```
btn1.Click += new System.EventHandler(btn1_Click);
```

unde `btn1_Click` este numele unei funcţii scrisă de programator pentru tratarea evenimentului de click pe buton.

1.2 Crearea vizuală a unei aplicaţii Windows

O parte a muncii de scriere cod sursă depusă până acum poate fi făcută automat de mediul de programare, dacă vom crea vizual aplicaţia.

Dacă programatorul alege corect tipologia aplicației, Designer-ul desenează el forma, iar prin dragarea de către programator a unor controale din **ToolBox**, mediul declară obiectele corespunzătoare, le stabileşte proprietățile şi le înregistrează la nivelul formei.

Sub biblioteca de clase MicroSoft Foundation Classes, **vizualizările** erau alese în funcție de specificul aplicației:

- *Forms*: formular – specializată în introducere date în machetă document
- *TreeView*: arborescentă – pentru vizualizări structuri expandabile de date
- *EditView*: editare - specializată în lucru cu texte
- *View* – vizualizare generică, pentru reprezentări grafice.

Sub .NET se revine aşadar la aplicații cu vizualizare unică, de tip **forms**, peste care se pun controale de vizualizare specializate (TreeView, ListView, DataGrid, TextBox, Panel etc.)

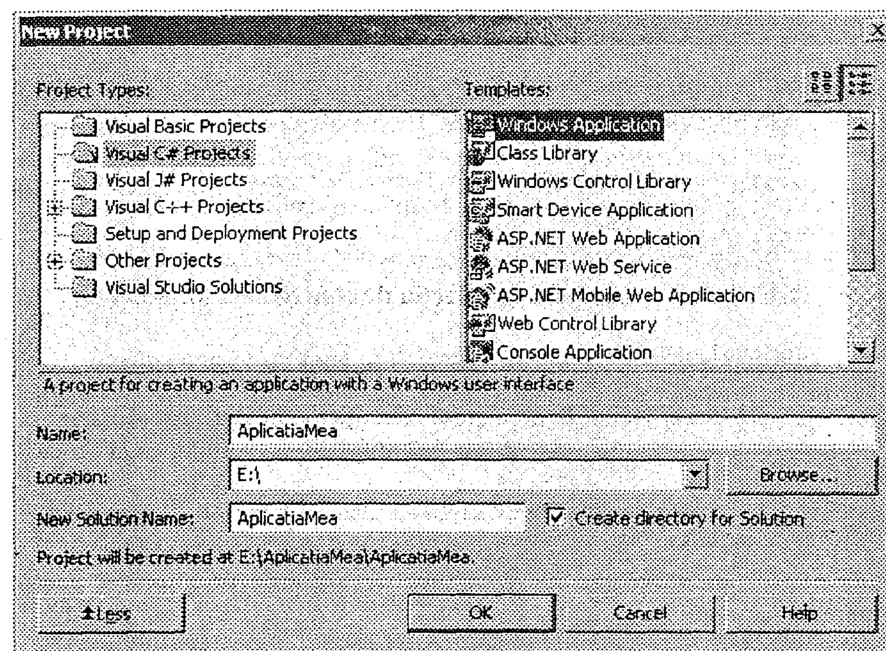


Fig. 3.2 Alegerea şablonului de aplicație

`System.Windows.Forms` este namespace-ul ce conține definiții pentru toate clasele uzuale de tip control şi funcțiile necesare lucrului cu formulare.

Există şabloane tip pentru aplicații (fig. 3.3); pentru o aplicație **Visual C#** sub Windows se alege din meniu:

File / New / Project Type: Visual C# Projects
Template: Windows Application

şi se dă un nume, împreună cu localizarea, pentru noua aplicație.

După alegerea şablonului de aplicație, mediul integrat crează scheletul aplicației. Lucrul poate continua apoi tot în manieră vizuală, mediul disponibilizând o serie de instrumente de programare vizuală.

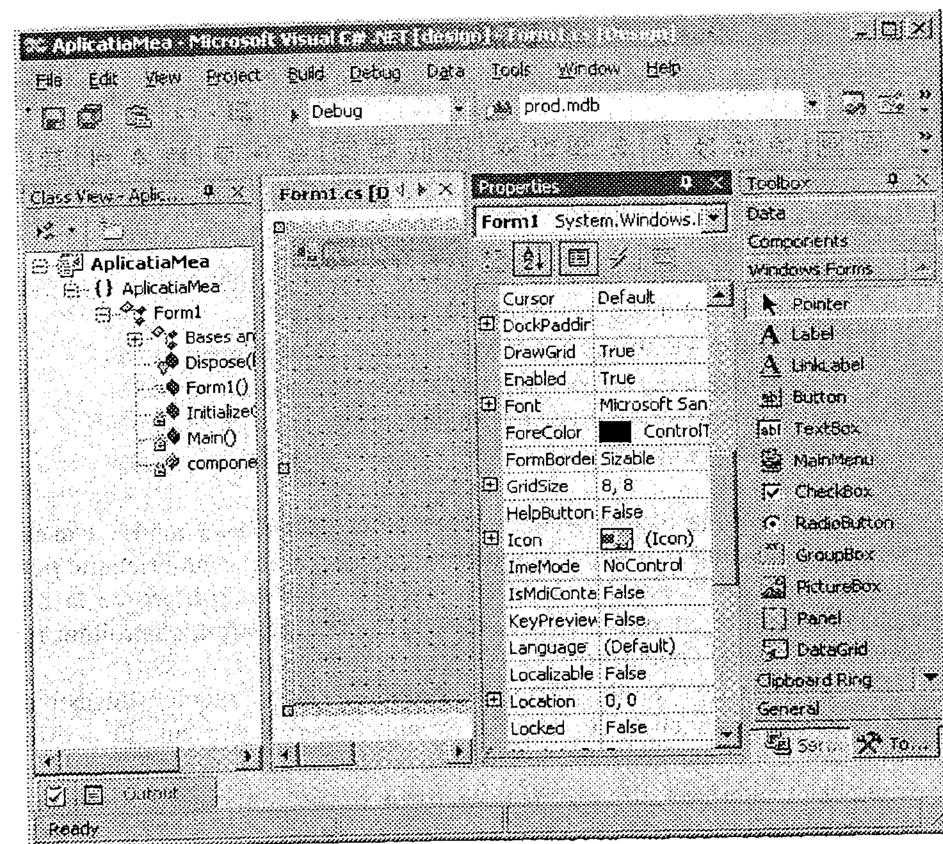


Fig. 3.3. Principalele ferestre utilizate în programarea vizuală

Ferestrele mai importante disponibilizate de către mediul Visual C# şi accesibile din meniul **View** sunt:

- **Class View** – care afişează structura pe clase a aplicaţiei şi permite accesul rapid la o funcţie din cadrul aplicaţiei;
- **ToolBox** - trusa cu controalele disponibilizate de mediu sau definite de utilizator şi utilizabile în aplicaţii;
- **Designer** – componenta ce afişează partea vizuală (resursele grafice ale aplicaţiei) şi scrie cod sursă pe măsură ce dezvoltăm vizual aplicaţia.
- **Code Xxxxx.cs** – fereastra ce conţine codul sursă al aplicaţiei, parţial compus de către designer pe măsura proiectării vizuale a aplicaţiei, parţial scris de programator.

- **Properties** – conţine proprietăţile obiectului curent selectat în Designer sau din codul sursă. Are două secţiuni: **Properties** - adică proprietăţile propriu-zise, sortate pe categorii sau alfabetic şi **Events** - conţinând evenimentele recunoscute şi/sau tratate de un control.

Proprietăţile mai importante ale obiectului `Form` sunt:

- `Name` – numele dat controlului de tip form ;
- `Text` – textul ce apare în bara de titlu a aplicaţiei;
- `BackgroundImage` – imaginea afişată pe fundal;
- `BackColor` – culoarea de fundal;
- `Font` – fontul de scriere etc.

Putem pune o proprietate (de exemplu culoarea de fond) a unor controale selectând mai multe controale simultan; nu toate proprietăţile pot fi setate însă în această manieră, deoarece ele trebuie să difere de la un control la altul (spre exemplu, `Location` – conţinând poziţia controlului în cadrul ferestrei de vizualizare).

Proprietatea `Name` permite atribuirea unor nume sugestive controalelor, iar funcţiile de tratare a evenimentelor recunoscute de aceste controale, când sunt generate automat au în denumire şi acest nume (de exemplu, `btnCalcul_Click`, sau `menSave_Click`), mărind lizibilitatea programelor.

Forma este denumită implicit `Form1`, iar funcţia *Main* lansează aplicaţia invocând constructorul formei, făcând astfel legătura între aplicaţie şi fereastra de rulare:


```
Application.Run(new Form1());
```

Dacă dorim să-i dăm un nume mai sugestiv, modificăm în Properties numele formei (`Name: Forma_mea`), apoi cu **Find / Replace** facem substituirea `Form1` cu `Forma_mea` şi în codul sursă. Când facem substituiri, trebuie să avem grijă să fie expandată şi funcţia `InitializeComponent()`, pentru a substitui eventualele apariţii şi în această zonă.

În C#, în locul funcţiilor independente se folosesc funcţii membre statice în clase; ele pot fi lansate în execuţie chiar înainte de a genera obiecte din clasa respectivă. Funcţia `Main()` este şi ea membră într-o clasă; în cazul aplicaţiilor formular ea face parte din clasa `Form1`.

Modificările făcute în *Designer* sunt operate automat în codul sursă; nu sunt indicate modificări manuale în zona de cod gestionată de către Designer.

> `#region` şi `#endregion` `sunt` directive pentru controlul expandării – comprimării unei zone din textul sursă, în timpul editării.

Adăugarea unei funcţii într-o clasă

- meniu **View / Class View**
- selectăm clasa dorită
- MouseRight şi alegem **Add Method**
- completăm caracteristicile dorite, apoi codul sursă al funcţiei.

Adăugarea vizuală a unei funcţii de tratare a evenimentelor

- meniu **View / Designer;**
- selectăm controlul care produce evenimentul; poate fi chiar forma principală a aplicaţiei;
- MouseRight + **Properties,** secţiunea **Events** (butonul)
- selectăm evenimentul dorit şi facem double click sau punem numele ales pentru funcţie;
- suntem introduşi în editorul de text sursă, unde completăm cu codul dorit;

- automat, mediul crează un delegat (referință de funcție), îl încarcă cu referința funcției nou create și îl adaugă pentru subscriere la evenimentul dorit:

```
this.button_Ad.Click +=
            new System.EventHandler(this.Aduna);
```

Dacă dorim să **renunțăm la o funcție de tratare** trebuie pus spațiu în **Properties / Events** pentru toate controalele care o invocă pe post de *handler* și eventual ștergem și fizic textul sursă al funcției; atenție căci de multe ori textul sursă este șters automat !

Exercițiu

Să se proiecteze vizual și să se scrie codul sursă necesar unei aplicații care **adună două numere** și afișează rezultatul.

Rezolvare

1. **File / New / Project Type: VC#**
   Template: Windows Application
2. Se aduc din **ToolBox** trei controale de tip *TextBox* ce vor conține numerele și trei controale de tip *Label* care vor eticheta (**Properties / Text**) cu **a, b** și **suma,** câmpurile anterioare. Se poate construi mai întâi un cuplu label + textBox, apoi selectăm cu shift + mouse cele două controale pe care apoi le multiplicăm prin Copy / Paste (Ctrl + C și Ctrl+V). Pentru a dimesiona pe verticală un textbox trebuie să-i declarăm proprietatea *Multiline* ca *true*.
3. Se colorează (**Properties / BackColor** + **Custom** sau **System**) în verde textbox-urile **a** și **b**, respectiv cu roșu câmpul **suma**.
4. Se separă cu o linie (GroupBox din ToolBox, micșorat la maxim, cu spațiu în proprietatea Text) primele două textbox-uri care sunt de introducere, de cel de-al treilea, care este destinat afișării.
5. Se denumesc textbox-urile (**Properties / Name**) cu **a, b** și **suma.**
6. Se aduc din ToolBox două controale de tip *Button*, ce vor fi denumite (**Properties / Name**) `b_Total` și `b_Iesire` și vor fi inscripționate (**Properties / Text**) **Total**, respectiv **Iesire.**

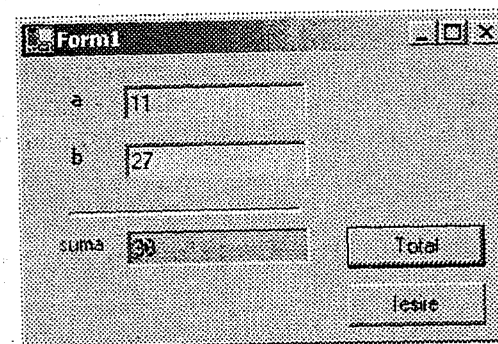
7. Pe butonul `Iesire` se pune (**Properties + Events / Click**) o funcție de tratare eveniment `Click`, conținând un apel de terminare a aplicației:


```
private void b_Iesire_Click
        (object sender, System.EventArgs e )
{
        Application.Exit();
}
```

Se putea și mai simplu, dând *double click* pe butonul `Iesire`; suntem astfel direct introduși în codul sursă pentru a completa corpul funcției de tratare `b_Iesire_Click()`, deoarece evenimentul *Click* de mouse este evenimentul implicit, de tratat pentru un buton.

8. Pe butonul **Total** se pune (**Properties + Events / Click**) o funcție de conversie a celor două numere din text în *int* și de adunare a lor, respectiv conversia rezultatului în text și vizualizarea lui ca proprietate `Text` a controlului **suma**.

```
private void b_Total_Click
        (object sender, System.EventArgs e)
{
        int i_a = Convert.ToInt32(a.Text),
            i_b = Convert.ToInt32(b.Text);
        suma.Text =(i_a + i_b).ToString();
}
```



Gestiunea unitară a controalelor unei forme

Uneori este nevoie de gestiunea unitară și folosirea în instrucțiuni repetitive, a controalelor; spre exemplu, tratarea similară a câmpurilor dintr-o factură, pentru a le inițializa sau pentru a calcula valoarea totală a facturii. În acest caz, se recomandă **gestiunea TextBox-urilor cu ajutorul masivelor de referințe**. Pentru a ne menține în spiritul limbajului C# și fără

a restrânge din generalitate, vom exemplifica acest lucru printr-un singur vector de referinţe.

1. **Declararea vectorului de referinţe** la TextBox, în definiţia clasei formular:

```
public System.Windows.Forms.TextBox []vr;
```

2. **Iniţializarea** în constructorul clasei formular, a elementelor **vectorului de referinţe**, cu obiecte generate dinamic sau cu referinţe ale unor obiecte care deja există:

```
vr = new System.Windows.Forms.TextBox [10];
vr[0]=textBox1;  vr[1]=textBox2;
          // controale deja existente pe forma

for(int i=2;i<5;i++)
   {
     vr[i]=new TextBox();
          // controale noi, create dinamic
     vr[i].Location =
          new System.Drawing.Point(16, 50+30*i);
     vr[i].TabIndex = i;
     vr[i].Text = 10*i+" ";
     this.Controls.Add(this.vr[i]);
   }
```

3. **Manipularea** obiectelor prin vectorul de referinţe, în funcţiile de tratare. Pentru testarea referirii corecte, la apăsarea unui buton Numerotare, fiecare TextBox îşi afişează (ca proprietate Text) propria poziţie.

```
private void Numerotare_Click(object sender, EventArgs e)
{
   for(int i=0;i<5;i++)     this.vr[i].Text=i+" ";
}
```

Exerciţiu

Să se construiască **macheta şi programul de completare a datelor într-un document,** conţinând mai multe linii şi coloane, câmpuri introductibile şi câmpuri calculate, precum şi părţi tipizate ale documentului. Se va asigura şi **serializarea documentului** (doar datele primare, nu şi cele calculate), cu posibilitatea **restaurării documentului** şi completarea automată a câmpurilor calculabile.

Rezolvare

Aplicaţia **FACTURA** exemplifică gestiunea prin matrice de referinţe a tuturor textBox-urilor dintr-un formular.

- se pune `TextBox [][] m;` la nivel de formă şi se alocă astfel referinţa la matricea de referinţe;
- în constructor se alocă vectorul de referinţe, apoi fiecare element din vector se va încărca cu adresa câte unui textBox, dintre cele create vizual (atenţie la ordinea în care au fost create):

```
m = new TextBox[3][];
m[0]= new TextBox[7]
      { textBox1,textBox2,textBox3,textBox4,
                     textBox5,textBox6,textBox7 };
m[1]=new TextBox[7]
    {  textBox14,textBox13,textBox12,textBox11,
                     textBox10,textBox9,textBox8 };
m[2]=new TextBox[7]
    {  textBox21,textBox20,textBox19,textBox18,
                     textBox17,textBox16,textBox15 };
```

- Se poate face o funcţie apelabilă din constructorul formei, pentru a testa legarea corectă a referinţelor la câmpuri; ea afişează linia şi coloana fiecărui câmp:

```
public void Umple_Forma()
{
   int i,j;
   for(i=0;i<3;i++)
         for(j=0;j<7;j++)
                m[i][j].Text=i.ToString()+j.ToString();
}
```

Ea poate fi apoi **refolosită** la iniţializarea cu spaţii sau zero a formei, la începutul lucrului sau după o salvare pe disc a unei facturi şi pregătirea unei noi completări.

În antetul programului este nevoie de două *namespace*-urile ce cuprind prototipurile claselor şi metodelor necesare serializării.

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

Pentru a beneficia de serializare se defineşte o clasă ale cărei obiecte sunt linii din factură (doar câmpurile introduse, nu şi cele calculate); ea poartă atributul de "serializabilă":

```
[Serializable]
class LinFact
{
        public string den;
        public string cant;
        public string pret;
        public LinFact(string d,string c, string p)
                { den=d; cant=c; pret=p;}
}

private void Salvare_Click(object sender, System.EventArgs e)
{
        ArrayList list = new ArrayList();
        for(int i=0;i<m.Length;i++)
        {
                LinFact lin = new LinFact
                        ( m[i][1].Text,m[i][2].Text,m[i][3].Text );
                list.Add(lin);
        }
        FileStream s =
                new FileStream("factura.dat",FileMode.Create);
        BinaryFormatter f = new BinaryFormatter();
        f.Serialize(s, list);
        s.Close();
        // de adaugat o functie ClearForm()si apelul ei aici
}

private void Restaurare_Click
                        (object sender, System.EventArgs e)
        {
                FileStream s;
                s = new FileStream("factura.dat", FileMode.Open);
                BinaryFormatter f = new BinaryFormatter();
                ArrayList lista = (ArrayList)f.Deserialize(s);
                s.Close();int i=0;

                foreach ( LinFact lin in lista)
                {
                 m[i][1].Text=lin.den;  m[i][2].Text=lin.cant;
                 m[i][3].Text=lin.pret;       i++;
                }

                Calcul_Click
                        ( this.button1, new System.EventArgs() );
        }
```


Form1

Nr_crt	Den_prod	Cant	Pret_unit	Val	Val_TVA	Val_Totala
1	Solutie de lipit	1	2	2	0.38	2.38
2	Creioane colorate	3	4	12	2.28	14.28
3	Caiet dictando tip 4	5	6	30	5.7	35.7
						52.36

Calcul · Salvare · Restaurare

Calculul se poate efectua simplu, cu o funcţie de genul următor:

```
private void Calcul_Click(object sender, System.EventArgs e)
{
        double p,c,total=0;
        for(int i=0;i<3;i++)
        {
                m[i][0].Text=(i+1).ToString();
                c=Convert.ToDouble(m[i][2].Text);
                p=Convert.ToDouble(m[i][3].Text);
                m[i][4].Text=(c*p).ToString();
                m[i][5].Text=(c*p*0.19).ToString();
                m[i][6].Text=(c*p*1.19).ToString();
                total+=c*p*1.19;
        }
        txtTotal.Text=total.ToString();
}
```

Lăsăm ca exerciţiu, modificarea programului de mai sus pentru a răspunde unor cerinţe precum, **machetarea unui document cu număr variabil de linii**, stabilit la fiecare rulare şi defilare în document folosind *scrollbar*. Se va desena un singur rând din factură, apoi celelalte rânduri vor fi create dinamic, în constructorul clasei *Form1*, de aceeaşi dimensiune cu celelalte şi plasate la poziţii calculate în funcţie de poziţia textBox-urilor din primul rând şi dimensiunea acestora. O altă modalitate ar fi să creem macheta cu toate liniile şi să ascundem (**Properties /** `Visible = false`) pe cele care nu ne interesează.

Cu mouse right pe ToolBox se poate alege opţiunea **`Sort Items Alphabetically`**, pentru a regăsi mai uşor un control, după nume.

Tehnici de lucru uzuale

- selectarea mai multor controale + **Copy / Paste** pentru obținerea de controale cu proprietăți similare;
- selectarea mai multor controale pentru alinierea lor sau pentru stabilirea unor proprietăți comune, inclusiv dimensiune;
- se poate pune dintr-odată aceeași funcție de tratare eveniment pe mai multe controale, dacă toate sunt selectate în momentul adăugării funcției .
- Dacă tastăm **F1** când suntem poziționați pe o eroare de compilare (în fereastra de *output*) se apelează automat help-ul adecvat erorii.
- Codul erorii poate fi folosit in *Search* pt a găsi detaliile despre eroare.
- Dacă zona din program, scrisă de Designer nu este expandată, **Find/Replace** nu o vede și nu va face substituirile dorite și în această porțiune. Nemodificând și numele folosit la += pentru delegat pentru o metodă al cărei nume dorim să-l schimbăm, se generează eroare de compilare.

Disponibilitatea unui control pentru utilizator de a interacționa cu acesta, când controlul este vizibil (`Visible` este `true`) se stabilește prin proprietatea `Enabled`, `true` sau `false`.

Controalele care pot primi focus, pot fi incluse într-o secvență de tab; se pot numerota rapid cu **View / TabOrder**, dând click în succesiunea de parcurgere dorită. Unele controale pot fi sărite din secvență dacă au proprietatea `TabStop` pe `false`. Dacă proprietatea `TabOrder` este `true`, se poate modifica succesiunea și static, doar pentru un control, schimbând valoarea din proprietatea `TabIndex`

Controalele au poziții relative la zona client a formei, în timp ce poziția formei este relativă la coordonatele ecranului.

Atașarea prin program a unui icon și a unei imagini de fundal pentru formă se poate face în constructor sau la încărcarea formei:

```
private void Form1_Load(object sender,  System.EventArgs e)
  {
    System.Drawing.Icon IC = new System.Drawing.Icon
                      ("C:\\Programs\\alarm1.ico");
    this.Icon = IC;
      Bitmap BG = new Bitmap("C:\\Programs\\poza.bmp");
```



```
      this.BackgroundImage = BG;
  }
```

C:\Program Files\Microsoft Visual Studio .NET 2003 \Common7\ Graphics \bitmaps\OffCtlBr\Small\Color
conține bmp-uri standard, utile în aplicații

Prin proprietatea `WindowState` se stabilește cum să apară forma la începutul rulării: `Normal`, `Maximized` sau `Minimized`.

Controalele CheckBox și RadioButton

Controlul **CheckBox** permite selectarea uneia sau mai multor opțiuni dintr-un grup de opțiuni, în timp ce controlul **RadioButton** permite selectarea unei singure opțiuni dintr-un grup și se folosește pentru opțiuni mutual exclusive.

Prezentăm în continuare câteva din proprietățile și evenumentele mai importante ce se regăsesc la controalele **CheckBox** și **RadioButton.**

`Checked` – marchează prin true și false dacă CheckBox-ul este bifat sau nu;.

Proprietatea `CheckState` specifică starea controlului și poate fi una din valorile enumerării `CheckState`: `Checked`, `Unchecked`, sau `Indeterminate`

Proprietatea `ThreeState` - indicates dacă CheckBox-ul permite trei stări. Pe false, controlul poate fi adus în starea `Indeterminate` doar prin cod sursă, nu și prin interfața grafică a designer-ului.

Proprietatea `Checked` nu indică totdeauna corect starea controlului, deoarece pentru un control cu trei stări, valoarea `true` este returnată atât pentru `Checked`, cât și pentru `Indeterminate`. În consecință, se recomandă preluarea stării curente din proprietatea `CheckState`.

Proprietatea `Text` conține textul explicativ ce însoțește controlul de tip bifă.

Caseta bifei se plasează uzual în stânga textului explicativ ce o însoțește, dar pot fi alese prin setarea proprietății `CheckAlign` și pozițiile

TopLeft, **TopCenter**, **TopRight**, **MiddleRight**, **BottomRight**, **BottomCenter**, respectiv **BottomLeft**, poziţii ce se găsesc ca enumerări în enum **ContentAlignment**:

```
checkBox1.CheckAlign = ContentAlignment.MiddleRight;
```

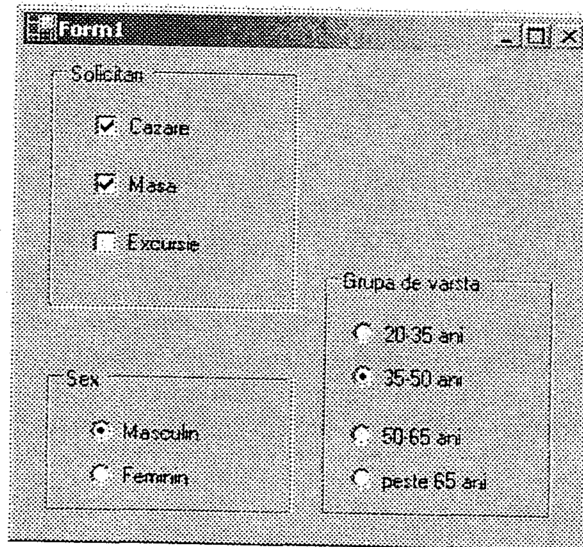
RadioButton şi CheckBox dispun de proprietatea **AutoCheck**; pusă pe true asigură ca la click pe ele să-şi schimbe automat starea vizuală, stare memorată şi în proprietăţile **Checked** şi **CheckState**.

Starea poate fi **setată** sau **comutată** şi soft prin atribuiri de genul:

```
this.checkBox1.Checked = true;
checkBox1.Checked = !checkBox1.Checked;
```

Pentru controalele **RadioButton** este importantă gruparea opţiunilor ce se exclud reciproc şi plasarea lor în containere diferite (**GroupBox** sau **Panel**).

CheckedChanged – este evenimentul implicit al controalelor **CheckBox** şi **RadioButton** şi se declanşează ori de câte ori starea controlului se schimbă.



Controlul Scroll Bar

Este o bară verticală (**VScrollBar**) sau orizontală (**HScrollBar**), dotată cu săgeţi şi cursor propriu pentru defilare. Ea asociază dimensiunea ei fizică cu un interval dat de valorile întregi, indicate prin proprietăţile **Minimum** şi **Maximum**. Poziţia curentă a cursorului ei se asociază dinamic cu valoarea proprietăţii **Value**. Modificând fizic la rulare, poziţia cursorului se modifică automat **Value**; invers, modificarea prin program a valorii din **Value** determină deplasarea fizică a cursorului pe poziţia echivalentă din lungimea barei.

Evenimentul implicit este **Scroll**, emis la orice defilare fizică în cadrul barii, folosind săgeţile sau cursorul. El poate fi captat şi folosit la obţinerea unor valori, preluând echivalentul din proprietatea **Value**.

Exemplu

- Se aduc din ToolBox un control VScrollBar, un TextBox şi un Button;
- Selectând controlul scrollBar1, se pun în **Properties**, valorile **1** şi **50** pentru **Minimum** şi **Maximum**, respectiv **1** şi **2** pentru proprietăţile **SmallChange** şi **LargeChange**, care controlează fineţea deplasării.
- cu DoubleClick pe scrollBar1, se înregistrează automat funcţia de tratare a evenimentului de **Scroll**, iar noi completăm doar conţinutul funcţiei, scriind:

```
textBox1.Text = vScrollBar1.Value.ToString();
```

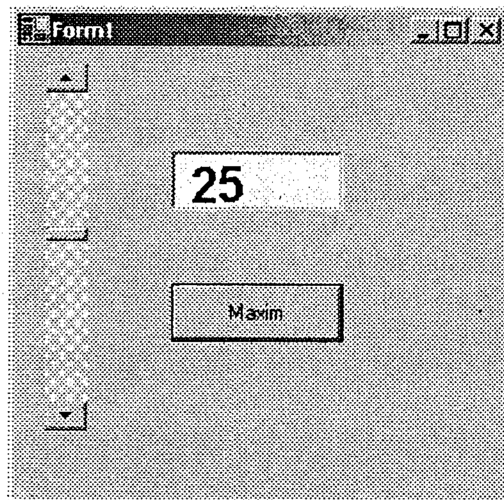
Aceasta face ca la o deplasare a cursorului, în TextBox să apară valoarea corespunzătoare din intervalul [1,50].

- Selectând şi dând DoubleClick pe buton se înregistrează automat evenimentul *Click* pe buton, iar noi completăm doar conţinutul funcţiei, scriind:

```
vScrollBar1.Value = 50;
```

pentru a testa şi operaţia inversă.

La rulare observăm că la deplasarea cursorului în cadrul barei, TextBox-ul afişează numărul echivalent poziţiei, iar la apăsare pe buton, cursorul este dus la maxim, adică echivalentul valorii 50.



Componenta ToolTip

Mecanismul de **ToolTip** permite afişarea unui text ajutător în legătură cu un obiect de interfaţă, atunci când mouse-ul rămâne în aşteptare mai mult timp deasupra obiectului. Deşi facilitatea pare strâns legată de obiectul cu care se asociază, ea este furnizată de către un control specializat, disponibil atât soft, cât şi vizual, în ToolBox: `ToolTip`.

Din punct de vedere al programării lucrurile sunt simple: se instanţiază un obiect al clasei `ToolTip`, iar metoda sa `SetToolTip()` permite asocierea textelor ajutătoare unor controale existente şi recunoscute la nivelul formei.

```
Form1()
 {
      /* ... */

  ToolTip t = new ToolTip();
  t.SetToolTip(b, "Buton creat soft");
  t.SetToolTip(statusBar1,
            "Aplicatia asteapta selectie optiune meniu");
  t.SetToolTip(toolBar1,
                  "Apasati butonul asociat actiunii dorite");
      /* ... */
 }
```

Nici **din punct de vedere vizual** lucrurile nu sunt prea complicate.

Unele controale (`TabPage`, `ToolBarButton`, `StatusBarPanel`) au nativ proprietatea **`ToolTipText`**, prin care se stabileşte textul de afişat ca tooltip.

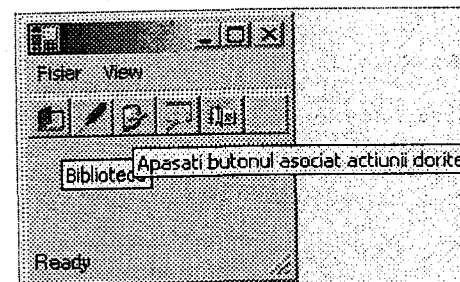
Pentru celelalte controale, aducând din ToolBox o componentă **`ToolTip`** vom observa că imediat **toate controalele de pe formă** vor dobândi o proprietate nouă: **`Tooltip On NumeComponenta`**, editabilă vizual şi prin intermediul căreia programatorul indică textul ajutător asociat controlului respectiv.

Acest lucru ne permite să deducem că putem folosi în paralel mecanismul soft şi cel vizual, sau să lucrăm cu mai multe componente `ToolTip` în acelaşi timp.

Spre exemplu, putem stabili un text la nivel de toolbar şi un altul, mai detaliat, la nivel de buton din toolbar:

```
this.toolBar1.ShowToolTips = true;
```

iar în colecţia `Buttons` a obiectului `toolBar1` se alege un buton şi i se completează proprietatea `ToolTipText` cu textul dorit.



2. Derivarea controalelor

Discutam mai devreme despre posibilitatea supraîncărcării (**`override`**) unor funcţii moştenite dintr-o clasă de bază. Am realizat deja acest lucru pentru funcţia `OnPaint` a clasei `Form`. Dacă privim mai atent, realizăm că nu putem face acelaşi lucru pentru oricare control deţinut de formă, pentru simplu motiv că nu putem supraîncărca funcţii decât în clasele derivate de noi din clasele de bază ale .NET Framework. Toate controalele de pe formă sunt instanţiate direct din clasele lor de bază, nu am definit noi clase derivate din acestea.

O primă necesitate de a deriva propriile noastre clase din clasele de bază rezidă aşadar în nevoia de a supraîncărca o funcţie (eventual de tratare a unui eveniment) prin propriul nostru cod sursă.

O a doua raţiune pentru care derivăm propriile noastre clase din clasele de bază este legată de nevoia de a induce comportamente particulare unor controale, în locul celor generale cu care sunt ele înzestrate. Vom exemplifica acest lucru în exerciţiul următor.

Exerciţiu

Să se deriveze din TextBox controlul **TextBoxNumeric** care nu acceptă în intrare decât cifre, celelalte caractere neglijându-le.

Rezolvare

Se descrie noua clasa **după încheierea definiţiei clasei Form1**, altfel Designerul n-ar mai şti care clasă asigură vizualizarea. În cadrul clasei se supraîncarcă funcţia `OnKeyPress`, care în mod uzual tratează evenimentul `KeyPress` asigurând înscrierea caracterelor introduse de la tastatură în câmpul afişat de TextBox.

```
public class TexBoxNumeric: TextBox
{
	protected override void OnKeyPress(KeyPressEventArgs e)
	{
		if(e.KeyChar >='0' && e.KeyChar <= '9' )
			base.OnKeyPress (e);
		else
			e.Handled=true;
	}
}
```

Pentru caracterele acceptate, se apelează forma originală (moştenită din bază), în timp ce pentru celelalte se "înghite" evenimentul anunţând că a fost tratat (`e.Handled=true`). Avem în acest fel şi un gen de validare a ceea ce introduce utilizatorul de la tastatură.

Pentru a testa comportamentul noii clase, într-o aplicaţie obişnuită se declară o referinţă la un obiect `TexBoxNumeric`:

```
	TexBoxNumeric txtBoxNum1;
```


iar în constructorul formei se instanţiază acest tip de obiect şi i se stabilesc proprietăţile de vizualizare, după care este adăugat la colecţia de controale a formei:

```
public Form1()
{
	InitializeComponent();
	txtBoxNum1= new TexBoxNumeric();
	txtBoxNum1.Location = new Point(100, 200);
	this.Controls.Add(txtBoxNum1);
}
```

Nu ne mai rămâne decât să ne convingem că TextBox-ul nostru lucrează bine, punând pe un buton o conversie în întreg a textului reţinut de control şi un calcul elementar (dublarea numărului citit):

```
private void btnCalcClick(object sender, System.EventArgs e)
{
  int x= Convert.ToInt32(this.txtBoxNum1.Text);
  this.txtBoxNum1.Text= (2*x).ToString();
}
```

3. Modificarea dinamică a proprietăţilor

O parte din proprietăţile obiectelor grafice pot fi nu numai iniţializate static, prin Designer, ci şi modificate dinamic, pe parcursul rulării. În **Properties / Dynamic Properties** putem vedea toate proprietăţile ce pot fi modificate, folosind denumirea lor sau nume alternative.

Pentru început vom exemplifica cele de mai sus, cu două funcţii care tratează evenimentele de `MouseEnter` şi `MouseLeave`, schimbând textul cu care este inscripţionat (caption) butonul `b_Iesire` dintr-o aplicaţie.

```
private void Iesire_MouseEnter(object sender,
                                        System.EventArgs e)
	{ b_Iesire.Text="Exit";    }

private void Iesire_MouseLeave(object sender,
                                        System.EventArgs e)
	{ b_Iesire.Text="Iesire";  }
```

Vom vedea acum cum prin modificarea dinamică a unor proprietăţi (vizibilitate, stare activă etc.) putem reconfigura controalele de pe formă pentru a ghida utilizatorul în introducerea datelor sau pentru a disponibiliza temporar noi facilităţi ale aplicaţiei în regim de lucru "administrator".

Exerciţiu

Să se proiecteze o machetă pentru o aplicaţie de atribuire a unor mărci persoanelor dintr-o firmă. Se vor folosi controale de tip **ComboBox**, **ListBox** din care se preiau marca şi numele unor persoane şi **ListView**, în care se vor vizualiza perechile marca-nume, deja formate.

Rezolvare

Toate cele trei controale au un element comun şi anume faptul că ţin mai multe item-uri într-o colecţie.

1. Într-o aplicaţie formular se adaugă din **ToolBox** controalele `ComboBox`, `ListBox` şi `ListView` şi se denumesc (**Properties / Name**) prin **cb_marca**, **lb_pers**, **listView1**.
2. Se populează iniţial, pentru exemplificare, `cb_marca` cu câteva mărci (selectând controlul + **Properties / Items** şi se adaugă în colecţie mărcile dorite, terminate fiecare cu Enter).
3. Se populează iniţial **lb_pers** cu câteva persoane, procedând ca mai sus.
4. În `InitializeComponent()` se inhibă iniţial posibilitatea selectării unor nume de persoane înainte de extragerea unei mărci din combo.

```
this.lb_pers.Enabled = false;
```

5. Se configurează soft, în constructorul formei, vizualizarea de tip `ListView` cu două coloane, specificând antetul afişat, dimensiunea şi tipul de aliniere dorit:

```
listView1.Columns.Add
            ("Marca",80,HorizontalAlignment.Center);
listView1.Columns.Add
   ("Numele si prenumele",200,HorizontalAlignment.Left);
```

6. Aducem din ToolBox un buton **Adaug**, lângă `cb_marca` pentru a permite şi adăugarea de noi mărci, în afara celor introduse iniţial.

7. Pe evenimentul click al acestuia punem cod sursă de adăugare în combo a mărcii înscrisă în zona de editare:

```
private void Adaug_Click
                  (object sender, System.EventArgs e)
{
      cb_marca.Items.Add(cb_marca.Text);
}
```

8. Se trage din ToolBox un mainMenu şi i se pune opţiunea **Men_AdBut** care ascunde / repune butonul de adăugare, dacă dorim să inhibăm / activăm posibilitatea de adăugare mărci. Opţiunii din meniu i se tratează evenimentul *click* prin funcţia:

```
private void Men_AdBut_Click
                  (object sender, System.EventArgs e)
{
      if(Adaug.Visible==false) Adaug.Show();
       else                    Adaug.Hide();
}
```

9. La selectarea unei mărci din combo, **se inhibă controlul cb_marca şi se reactivează lista de persoane lb_pers**, pentru a putea completa o pereche marcă-nume:

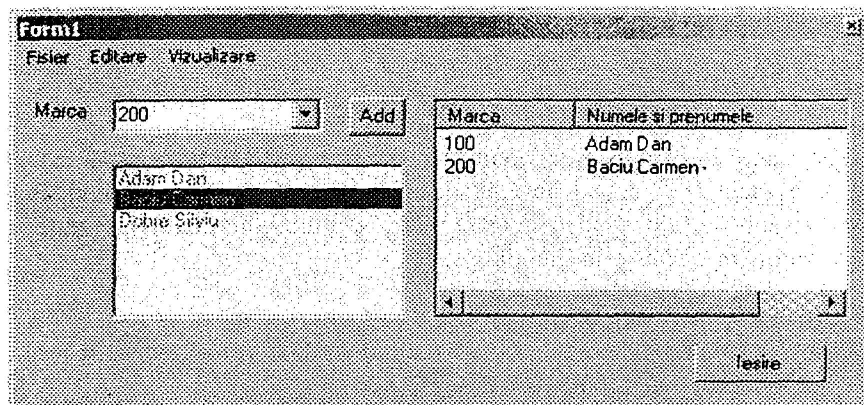
```
private void marca_SelectionChangeCommitted
                  (object sender, System.EventArgs e)
{
      this.lb_pers.Enabled=true;
      this.cb_marca.Enabled=false;

}
```

10. După selectarea şi a unei persoane, perechea constituită este pusă în lista de vizualizare folosind o structură de tip **ListViewItem**, după care se inversează stările celor două controale, pentru a permite reluarea procesului de atribuire mărci:

```
private void pers_SelectedIndexChanged
                  (object sender, System.EventArgs e)
{
   ListViewItem itm =
                  new ListViewItem(cb_marca.Text);
   itm.SubItems.Add(lb_pers.Text);
   listView1.Items.Add(itm);
```

```
        this.lb_pers.Enabled=false;
        this.cb_marca.Enabled=true;
    }
```



Să se dezvolte exemplul de mai sus cu facilitatea de adăugare de noi persoane în ListBox, precum şi cu tratarea opţiunilor de salvare / restaurare în/din fişier a perechilor marcă – nume deja formate.

4. Utilizarea indexărilor

Indexările sunt un gen de **supraîncărcări ale operatorului de indexare []** , dar nu apar ca metode, adică nu au operatorul ( ), ca orice funcţie, ci dau parametru direct între parantezele de indexare [];

Indexările permit adresare vectorială pe tipuri care nu sunt vectori (ex. bitul al 5-lea din reprezentarea binară a unui întreg); din această cauză, un tip înzestrat cu o indexare se mai numeşte şi "**smart array**";

Posedând accesori **`get`** şi **`set`**, indexările apar ca dezvoltări ale conceptului de vector, tot aşa cum proprietăţile sunt dezvoltări ale ideii de câmp.

Exerciţiu

Să se definească **tipul `biti`** prin înzestrarea tipului **`int`** cu capacitatea de autoindexare, pentru individualizarea unui bit după poziţia sa. Să se folosească acest tip pentru a memora prezenţa sau absenţa de la serviciu a unei persoane, în cele max. 31 de zile ale unei luni.

Rezolvare

Într-o aplicaţie C# de tip **Console Application** se introduce textul sursă:

```
using System;
namespace indexari
{

 struct Biti
  {
       private int b;
       public Biti(int val) { b = val;}

       public bool this[int idx]
        {
             get { return (b & (1<<idx))!=0; }
             set
             {
                  if(value) b |= (1<<idx);
                  else b &= ~(1<<idx);
             }
        }

  }

class Test
 {
      static void Main(string[] args)
       {
        Biti prezenta = new Biti(0);
        prezenta[5]=true; prezenta[27]=false;
        prezenta[7]=prezenta[5];
        Console.Write("Prezenta ziua de 7: " +prezenta[7] );
       }
  }
}
```

Se observă că întregul **`b`** conţinut în structura **`Biti`** este prelucrat folosind operatori pe biţi, pentru a seta, respectiv a extrage, doar bitul indicat prin index.

Indexarea **`public bool this[int idx]`** prin accesorii ei, **`get`** şi **`set`**, simplifică adresarea fiecărui bit, ca şi cum **`prezenta`** ar fi un vector de biţi.

O indexare poate fi doar de tip *read*, doar de tip *write*, sau de tip *read / write*, în funcţie de ce accesori furnizează (doar **`get`**, doar **`set`**, sau ambii).

Aşa cum **operator[]** nu se supraîncarcă decât prin funcţie membră nestatică, **o indexare nu poate fi definită static**; de altfel şi cuvântul cheie *this* sugerează legarea indexării de un obiect curent, nu de clasă în genere.

Un dezavantaj al soluţiei propuse este că numărul maxim de biţi este 32, adică dimensiunea unui `int`; pentru un număr mai mare, trebuie alocaţi mai mulţi întregi şi anume `n/32`, unde *n* este numărul de biţi necesari. Iată cum ar arăta în acest caz clasa care ţine biţii şi introduce indexările pentru identificarea fiecărui bit:

```
using System;

namespace indexari
{
 class Biti
  {
      int[] b;
      int lung;
      public Biti(int lung)
      {
           if (lung < 0) throw new ArgumentException();
           b = new int[((lung - 1) >> 5) + 1];
           this.lung = lung;
      }
      public int Lung
      {
           get { return lung; }
      }
      public bool this[int idx]
      {
           get
           {
                if (idx < 0 || idx >= lung)
                {
                     throw new IndexOutOfRangeException();
                }
                return (b[idx >> 5] & 1 << idx) != 0;
           }
           set
           {
                if (idx < 0 || idx >= Lung)
                {throw new IndexOutOfRangeException();    }
                if (value)
                { b[idx >> 5] |= 1 << idx;      }
                else { b[idx >> 5] &= ~(1 << idx); }
           }
      }
}
```



```
class Test
 {
  static void Main(string[] args)
    {
      Biti stare = new Biti(100);
      stare[5]=true;     stare[97]=stare[5];
      Console.Write("Bit de stare poz 97: " +stare[97] );
      Console.Read();
    }
  }
}
```

Expresia `(lung - 1) >> 5) + 1` determină numărul de întregi din vector, făcând împărţirea întreagă a lungimii şirului de biţi la 32 = 2^5 . Indexarea definită în clasă permite adresare atât în *read*, cât şi în *write*, testând în acelaşi timp şi încadrarea în lungimea şirului de biţi definită prin constructor.

Indexările pot fi definite, cum este şi firesc, pe vectori; în acest caz indexarea poate asigura regăsirea prin index a unui element.

Exerciţiu

Să se definească două **indexări pe un vector de persoane**, care să permită regăsirea unei persoane atât după nume, cât şi după marcă.

Rezolvare

Căutarea în vector se poate face cu metoda `Array.IndexOf(vp,p)`, definită în clasa `Array`; ea primeşte vectorul de persoane şi persoana căutată şi returnează poziţia elementului în vector, sau "-1" când elementul nu este găsit. Metoda se bazează pe funcţia `Equals(object altul)`, care testează că obiectul primit este de acelaşi tip cu obiectul curent şi în plus, îndeplineşte condiţia de egalitate specifică fiecărui tip de obiect.

Clasa `object` din care sunt derivate toate celelalte clase furnizează variante implicite pentru funcţiile `GetHashCode()`şi `Equals()`, dar obiectele particulare (clasa `Pers`, în exemplu nostru) trebuie să le redefinească, pentru a preciza cum se va genera codul de hashing, respectiv când două astfel de obiecte se consideră a fi egale.

În cazul nostru, **codul de indexare** s-a obţinut pornind de la `Marca`, iar la căutare două persoane se consideră "egale" dacă fie mărcile, fie numele lor corespund. În felul acesta putem căuta o persoana fie după marcă, fie după nume.

De data aceasta, aplicaţia va fi construită ca aplicaţie Windows, nu de tip consolă. Pe o formă de vizualizare au fost puse două textbox-uri, unul pentru **Marca**, altul pentru **Nume** şi două butoane, pentru **Adaugare**, respectiv **Cautare**.

Pornind de la o clasă **Pers**, dotată cu un constructor specific, s-a definit o altă clasă BdPers, care în esenţă este **un vector de obiecte** de tip Pers. Ea aduce în schimb, două indexări, una după nume şi alta după marcă. Aşadar **pot fi date mai multe indexări, dacă ele diferă prin prototip** (în cazul nostru, una primeşte *int*, alta *string*) .

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace indexPers
{
public class Form1 : System.Windows.Forms.Form
{
 private System.Windows.Forms.Button Adaug;
 private System.Windows.Forms.TextBox tb_Marca;
 private System.Windows.Forms.TextBox tb_Nume;
 private System.Windows.Forms.Button Cauta;

 private System.ComponentModel.Container components = null;

 public Form1()
  {
      InitializeComponent();
      BDP=new BdPers();
  }

 protected override void Dispose( bool disposing )
  {
      if( disposing )
       {
            if (components != null)
            {
                  components.Dispose();
            }
       }
      base.Dispose( disposing );
  }

#region Windows Form Designer generated code
```



```
private void InitializeComponent()
 {
      this.tb_Marca = new System.Windows.Forms.TextBox();
      this.tb_Nume = new System.Windows.Forms.TextBox();
      this.Adaug = new System.Windows.Forms.Button();
      this.Cauta = new System.Windows.Forms.Button();
      this.SuspendLayout();
      //
      // tb_Marca
      //
      tb_Marca.Location = new System.Drawing.Point(24, 32);
      tb_Marca.Name = "tb_Marca";
      tb_Marca.Size = new System.Drawing.Size(80, 20);
      this.tb_Marca.TabIndex = 0;
      this.tb_Marca.Text = "tb_Marca";
      //
      // tb_Nume
      //
      tb_Nume.Location = new System.Drawing.Point(152, 32);
      this.tb_Nume.Name = "tb_Nume";
      this.tb_Nume.Size = new System.Drawing.Size(208, 20);
      this.tb_Nume.TabIndex = 1;
      this.tb_Nume.Text = "tb_Nume";
      //
      // Adaug
      //
      this.Adaug.Location = new System.Drawing.Point(24, 64);
      this.Adaug.Name = "Adaug";
      this.Adaug.Size = new System.Drawing.Size(80, 23);
      this.Adaug.TabIndex = 2;
      this.Adaug.Text = "Adauga";
      this.Adaug.Click +=
                  new System.EventHandler(this.Adaug_Click);
      //
      // Cauta
      //
      Cauta.Location = new System.Drawing.Point(208, 64);
      this.Cauta.Name = "Cauta";
      this.Cauta.TabIndex = 3;
      this.Cauta.Text = "Cauta";
      this.Cauta.Click +=
            new System.EventHandler(this.Cauta_Click);
      //
      // Form1
      //
      AutoScaleBaseSize = new System.Drawing.Size(5, 13);
      ClientSize = new System.Drawing.Size(376, 109);
      this.Controls.Add(this.Cauta);
      this.Controls.Add(this.Adaug);
      this.Controls.Add(this.tb_Nume);
```

```
        this.Controls.Add(this.tb_Marca);
        this.Name = "Form1";
        this.Text = "Form1";
        this.ResumeLayout(false);

    }
 #endregion

 [STAThread]
 static void Main()
 {
        Application.Run(new Form1());
 }

 public BdPers BDP;

 private void Adaug_Click(object sender, System.EventArgs e)
 {
        int m=Convert.ToInt32(tb_Marca.Text,10);
        BDP.vp[BDP.crt]=new Pers(m,tb_Nume.Text); BDP.crt++;
 }

 private void Cauta_Click(object sender, System.EventArgs e)
 {
        Pers p;
        if( tb_Nume.Text=="" && tb_Marca.Text!="" )
                 p=BDP[Convert.ToInt32(tb_Marca.Text)];
        else if ( tb_Nume.Text!="" && tb_Marca.Text=="")
                 p=BDP[tb_Nume.Text];
           else
              {
               MessageBox.Show
                   ("Cautare dupa o sg caracteristica!");
               p=new Pers(0,"");
              }

        tb_Marca.Text=p.Marca.ToString(); tb_Nume.Text=p.Nume;
  }
}


public class Pers
{
  public  int Marca;
  public string Nume;
  public Pers(int m, string n){Marca=m; Nume=n;}

  public override int GetHashCode()
      { return Marca.GetHashCode(); }
```



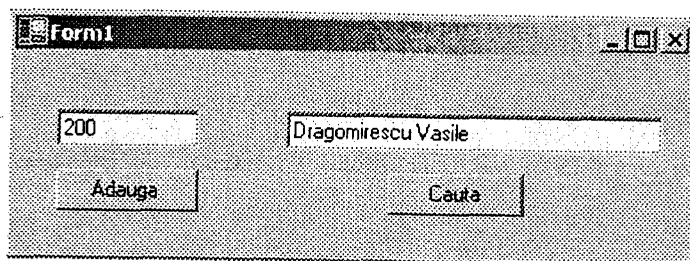
```
  public override bool Equals(object altul)
      { return (altul is Pers) && Equals((Pers) altul);}

  public bool Equals(Pers altul)
      {
        return Nume == altul.Nume || Marca== altul.Marca;
      }
 }

public class BdPers
 {
  public Pers [] vp;
  public int crt; int dim;
  public BdPers()
    {
      dim=100; crt=0;
      vp=new Pers [100];
    }
  public Pers this[int m]
   {
      get
       {
             Pers p=new Pers(m,"");
             int poz = Array.IndexOf(vp,p);
             return (poz != -1 ? vp[poz]:
                    new Pers(0,"<Anonim>")   );
       }
  }

  public Pers this[string n]
   {
      get
      {
             Pers p=new Pers(0,n);
             int poz = Array.IndexOf(vp,p);
             return (poz != -1 ? vp[poz]:
                    new Pers(0,"<Anonim>")   );
      }
   }
 }
}
```

Lucrând cu mai multe clase, trebuie să reamintim că `Form1` trebuie să fie declarată totdeauna ca primă clasă, pentru a fi prelucrabilă prin *Designer*.



Când o variabilă dintr-o clasă nu este "văzută" în editor la tastarea "." după numele obiectului, înseamnă probabil că este `private` şi deci inaccesibilă direct.

VALIDAREA DATELOR. GESTIUNEA ERORILOR ŞI EXCEPŢIILOR

1. Validarea datelor introduse într-un formular
2. Gestiunea excepţiilor

1. Validarea datelor introduse într-un formular

Am văzut deja cum utilizatorul poate fi ghidat să completeze un formular într-o anumită ordine a câmpurilor. Evident, se va pune şi problema validării conţinutului câmpurilor; când varietatea valorilor posibile pentru un câmp este redusă, controale precum `ListBox` sau `ComboBox` oferă o variantă destul de comodă de implementat. Când însă domeniul valorilor este foarte mare sau infinit, este nevoie de funcţii de validare specifice, uneori corelând valorile înscrise în mai multe câmpuri, spre exemplu şcolile absolvite corelate cu vârsta unei persoane.

Validare simplă

În subcapitolul privind **Gestiunea excepţiilor** vom vedea o modalitate standard de gestiune a situaţiilor neprevăzute şi a erorilor, aplicabilă şi erorilor de validare, bazată pe utilizarea blocurilor `try` şi `catch`. Aici însă, vom dezvolta şi alte modalităţi specifice de validare a datelor utilizate de o aplicaţie, folosind componente specializate şi proprietăţi ale obiectelor.

Controalele dispun de proprietăţi şi recunosc evenimente ce pot fi corelate direct cu validarea erorilor. Spre exemplu, punând pe **`true`** proprietatea **`CausesValidation`** a unui control sau a formei se solicită ca în momentul sosirii pe control să se declanşeze funcţia de validare asociată controlului care tocmai a pierdut focus-ul.

Validarea presupune în acest caz execuţia funcţiilor declarate pentru tratarea evenimentelor **`Validating`** şi **`Validated`**.

Evenimentul **`Validating`** se produce la încercarea de părăsire a unui control pentru a trece pe un alt control care are proprietatea **`CausesValidation`** pusă pe **`true`**.

Exerciţiu

Să se scrie funcţia de validare a vârstei unei persoane pentru încadrare în intervalul [1, 100].

Rezolvare

Pe un formular se adaugă un textbox **Varsta** şi un buton. Butonului i se pune proprietatea **CausesValidation** pe **true**, iar textbox-ului i se adaugă un handler pe evenimentul **Validating**:

```
private void Varsta_Validating
     (object sender, System.ComponentModel.CancelEventArgs e)
{
 string vs =( (TextBox)sender).Text;

 int vi =System.Convert.ToInt32(vs,10);
 if(vi<0 || vi>100)
  {
   e.Cancel = true;
   MessageBox.Show("Varsta trebuie sa fie intre 0-100",
      "Eroare",MessageBoxButtons.OK,MessageBoxIcon.Warning );
  }
}
```

La încercarea de părăsire a câmpului Varsta, când valoarea înscrisă nu aparţine intervalului fixat, evenimentul (indicat prin parametrul funcţiei) este revocat (**e.Cancel=true**;), preîntâmpinând trecerea focus-ului pe buton; cursorul este readus automat în câmpul vârsta pentru a introduce o valoare corectă.

Valoarea înscrisă în câmp a fost identificată mai general, pornind de la cast pe obiectul care a emis mesajul, dar putea fi simplu referită prin Varsta.Text.

Evenimentul **Validated** se declanşează după **Validating**, dar nu înainte de părăsirea controlului şi numai **dacă nu a fost revocat** evenimentul **Validating.**

Validarea simplă e utilă pentru testarea izolată, doar a unui control; pentru validare încrucişată acest mecanism e dificil de gestionat deoarece:

- nu ştiu unde se va duce utilizatorul, deci **CausesValidation** trebuie pusă pe true la toate celelalte controale, altfel efectul ar depinde de reacţia utilizatorului;
- nu acţionează şi pentru toolbar şi meniuri bară;
- în cazul validărilor mai complicate produce blocaje în lanţ, fiecare control cerând şi aşteptând să fie validat controlul precedent.

Validări încrucişate

Pentru că de cele mai multe ori validarea presupune corelarea valorilor mai multor câmpuri, acest lucru este imposibil de realizat la părăsirea unui câmp, deoarece nu ştim în ce ordine a preferat utilizatorul să introducă datele în câmpuri. De altfel, momentul validării trebuie atent stabilit, deorece pus prea devreme nu dispunem de toate datele introduse, necesare validărilor complexe, iar pus prea târziu, corectarea unei erori depistate ar însemna foarte multe câmpuri de reintrodus.

Ca test, vă propunem validarea încrucişată efectuată la tentativa de salvare a datelor, când se presupune că utilizatorul a introdus valori pentru toate câmpurile obligatorii.

Exerciţiu

Să se dezvolte exemplul de mai sus privind validarea vârstei unei persoane asigurând şi corelarea vârstei cu informaţia privind şcolile absolvite.

Rezolvare

- În general, ne asigurăm mai întâi că toate controalele şi forma au proprietatea **CausesValidation** pusă pe **false**, pentru a nu declanşa validări individuale, în buclă închisă.
- Pe butonul Salvare punem apelul funcţiei de validare, ce conţine toate clauzele dorite:

```
private void btnSalvare_Click(object sender, EventArgs e)
  {
      string vs = Varsta.Text;  string mes = "";
      bool err = false;
      int vi =System.Convert.ToInt32(vs,10);
      if(vi<0 || vi>100)
       {
        mes+="\nVarsta in afara intervalului 0-100";
        err=true;
       }
      if( vi<15 && Scoala.Text=="Facultate")
       {
           mes+="\nVarsta nu se coreleaza cu scoala";
           err=true;
       }
      if(!err) ;//... salvare date
       else MessageBox.Show(mes);
  }
```

O alternativă pentru cazul în care sunt multe erori de raportat, iar urmărirea lor ar fi greoaie, se bazează pe serviciile unui control de tip **ErrorProvider**, extras din ToolBox.

Spre exemplu, pentru corelarea vârstei cu ultima şcoala absolvită, se procedează astfel:

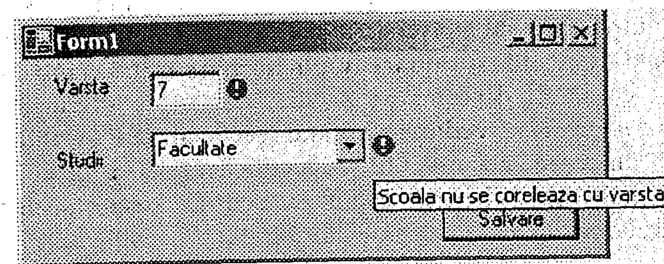
- se construiesc controalele de bază, un TextBox `Varsta` şi un ComboBox `Scoala`;
- se încarcă pentru ComboBox proprietatea **`Items`** / **`Collection`** cu valorile posibile (Elementara, Gimnaziu, Liceu, Facultate);
- se aduce în partea de jos a formei, un control **ErrorProvider**, căruia i se stabileşte proprietatea **`BlinkStyle`** pe valoarea **`BlinkIfDifferentError;`** beneficiind de tehnologia componentelor, el va putea anunţa celelalte controale de prezenţa sa;
- la aducerea acestui control, controalele `Varsta` şi `Scoala` vor afişa o proprietate în plus, **`Error on errorProvider`**, prin care se indică textul iniţial de afişat în cazul unei erori; stabilirea textului de afişat pe eroare are rolul de a marca cu un icon de avertisment toate controalele ce vor intra în procesul de validare. Ulterior, textul poate fi schimbat cu ajutorul metodei **`SetError`** aparţinând controlului **ErrorProvider;**
- controalele **`Varsta`** şi **`Scoala`** vor afişa chiar din faza de proiectare în designer, în dreapta lor, şi un icon de avertisment, care la momentul execuţiei va clipi la producerea unei noi erori de validare, iar la trecerea mouse-ului va afişa ca tooltip mesajul de eroare stocat în acel moment în proprietatea **`Error on errorProvider`** a controlului respectiv;
- pe butonul **`Salvare`** punem apelul funcţiei de validare, ce conţine toate clauzele dorite:

```
private void btnSalvare_Click(object sender, EventArgs e)
 {
  string vs = Varsta.Text;
  int vi =System.Convert.ToInt32(vs,10);
  if(vi<0 || vi>100)
      errorProvider1.SetError
          (Varsta,"Varsta in afara intervalului 0-100");
  else
      if( vi<15 && Scoala.Text=="Facultate")
       {
       errorProvider1.SetError
           ( Varsta,"Varsta nu se coreleaza cu scoala");
       errorProvider1.SetError
           ( Scoala,"Scoala nu se coreleaza cu varsta");
       }
      else
```



```
       { // .. .. salvare date
           errorProvider1.SetError(Varsta,"");
           errorProvider1.SetError(Scoala,"");
       }
 }
```

Când în proprietatea **`Error on errorProvider`** există mesaj `null`, nu se afişează nici icon-ul de avertisment, lucru care se întâmplă când nu s-a produs nici una din condiţiile de eroare.



La umplerea cu spaţii a numelui funcţiei apelate pe un eveniment se realizează atât *unsubscribe*, adică detaşarea handlerului de la evenimentul respectiv, cât şi ştergerea codului sursă al funcţiei, lucru care poate conduce la pierderea de text sursă.

2. Gestiunea excepţiilor

Soluţia obiectuală pentru gestiunea situaţiilor neprevăzute, de genul depăşirii memoriei alocate, încercării de a citi dintr-un fişier care nu mai există etc., este implementată prin intermediul **excepţiilor**. Limbajele de programare care au aderat la platforma .NET trebuie să implementeze şi gestiunea excepţiilor.

Blocurile `try` şi `catch`

În C# cea mai simplă modalitate de a gestiona o eroare constă în a grupa instrucţiunile suspectate că ar putea produce o eroare într-un bloc ***try***. Acest lucru sugerează o încercare supravegheată a execuţiei instrucţiunilor respective, captând eventualele erori şi tratându-le într-un bloc ***catch***, asociat blocului ***try***.

```
try
 {
       // instrucţiuni suspectate că ar putea produce excepţii
 }

catch
 {
       // instrucţiuni de tratare a eventualelor excepţii
 }
```

Excepţiile pot necesita tratamente diferenţiate în funcţie de tipul lor; spre exemplu o eroare de conversie a unui text în număr întreg ar putea fi tratată punând ca rezultat **o valoare implicită sau cerând reintroducerea** , în timp ce o eroare de depăşire a capacităţii de stocare a elementelor unui vector se poate rezolva **redimensionând vectorul sau oprind execuţia** programului. În consecinţă, este util să identificăm tipul erorii şi eventual să oferim mai multe blocuri ***catch*** conţinând cod de tratare distinct pentru fiecare tip de eroare în parte.

```
try
 {
       // instrucţiuni suspectate că ar putea produce excepţii
 }

catch(ExceptionType A)
 {
       // instrucţiuni de tratare excepţii tip A
 }

catch(ExceptionType B)
 {
       // instrucţiuni de tratare excepţii tip B
 }
```

Vom lua ca exemplu o aplicaţie Windows C# care împarte doi întregi, furnizând un rezultat întreg. Erorile cele mai frecvente se produc în zona de conversii şi calcul, astfel încât aceste instrucţiuni le grupăm într-un bloc `try`. Astfel, funcţia de tratare a apăsării pe butonul de calcul va putea fi descrisă în maniera următoare:

```
private void Divide_Click(object sender, System.EventArgs e)
{
 int a,b,r;
 try
  {
       a=int.Parse(txtA.Text); b= int.Parse(txtB.Text);
```



```
       r=a/b;
       txtR.Text=r.ToString();
  }

 catch(FormatException fmtExc)
  {
       txtR.Text="Eroare"; // afisare in TextBox
       MessageBox.Show(
             "Eroare in formatul de intrare. Reintroduceti!" ,
             fmtExc.Message);
  }
 catch(DivideByZeroException dvdExc)
  {
       txtR.Text="Eroare";// afisare in TextBox
       MessageBox.Show("Impartire la zero !",
       dvdExc.Message);

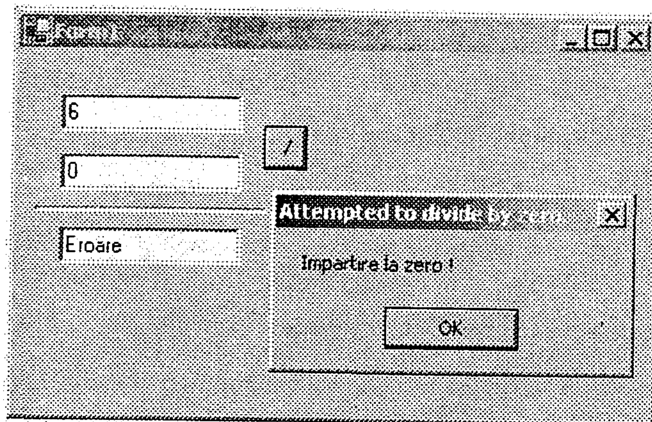
  }
}
```

Sunt captate două categorii de erori: **erori de conversie format** (clasa `FormatException`) şi **erori de împărţire la zero** (clasa `DivideByZeroException`), cărora le corespund blocuri `catch` distincte.

Observaţii.

- Declararea variabilelor `a, b, r` trebuie să se facă în afara blocului `try` dacă ele sunt folosite şi în afara blocului (adică în blocul `catch`).
- Operaţiile **în virgulă mobilă** nu ridică excepţii, ci le rezolvă intern, după următoarele reguli:
  - dacă valoarea rezultată **este prea mică** pentru formatul de reprezentare, ea este afişată ca `+0` sau `-0`;
  - dacă valoarea rezultată **este prea mare**, ea este afişată `Infinity`;
  - dacă operaţia cerută **este ilegală** se afişează `NaN` (Not **a** Number).

Biblioteca de clase .NET oferă mai multe clase asociate diferitelor tipuri de excepţii; de obicei excepţiile sunt organizate în ierarhii, pe baza derivării unora din altele. Spre exemplu, excepţia `DivideByZeroException` este derivată din `ArithmeticException`, care la rândul ei este derivată din `SystemException`, derivată în ultimă instanţă din clasa generică `Exception`.



Din punctul de vedere al bibliotecii de clase, erorile sunt împărţite, pe cel mai înalt nivel, în două categorii:

- `ApplicationException` – **excepţii de aplicaţie**, generate de aplicaţiile utilizator;
- `SystemException` - **excepţii de sistem**, generate de către maşina virtuală, Common Language Runtime.

Ambele sunt derivate din clasa generică `Exception`. Din punctul de vedere al limbajului, `Exception` este un obiect care încapsulează informaţii despre eroare întâlnită, informaţii care ar putea ajuta la tratarea erorii.

Încercarea de rezolvare a unei excepţii se bazează pe **specificitatea erorii**, adică se caută un bloc `catch` specific erorii produse şi numai dacă acesta nu există se caută blocuri `catch` asociate excepţiilor plasate din ce în ce mai sus în ierarhia de derivare. Din această cauză **se recomandă şi plasarea blocurilor `catch` în ordinea de la erori specifice către erori din ce în ce mai generale.**

Când excepţia se produce în interiorul mai multor **apeluri succesive de funcţii,** netratarea excepţiei în apelant declanşează propagarea ei în sus către apelant şi aşa mai departe până la nivelul cel mai înalt, Common Language Runtime, care presupune implicit oprirea execuţiei aplicaţiei.

Spre exemplu un bloc `catch` de forma celui de mai jos tratează orice excepţie conformă cu Common Language Specification:

```
catch(Exception exc)
{
 string msg =
    String.Format("Message: {0}\n Stack Trace:\n {1}",
    exc.Message, exc.StackTrace );
```



```
    MessageBox.Show(msg, exc.GetType().ToString());
}
```

în timp ce un bloc catch fără parametri tratează orice eroare chiar neconformă cu specificaţia limbajelor care au aderat la platforma .NET:

```
catch
{
      // tratare exceptii în general ;
}
```

Un bloc `try` trebuie să aibă asociat cel puţin un bloc `catch`; dacă nu, el trebuie măcar să se asocieze cu un bloc **`finally`**, care să-i motiveze utilitatea.

Blocul `finally`

Blocul **`finally`** este unic, spre deosebire de `catch` care poate fi multiplu. El descrie operaţiile ce trebuie să se execute îndiferent dacă o excepţie s-a produs sau nu şi urmăreşte aducerea la o stare coerentă şi consistentă a aplicaţiei, indiferent de succesul sau insuccesul acţiunilor din `try` şi eventualele blocuri `catch`.

Spre exemplu, eşuarea alocării integrale a unui pachet de resurse nu trebuie să lase ca ocupate resursele ce au fost deja alocate.

```
try
{
      //  instrucţiuni ce fac alocări resurse
}

finally
{
      // eliberare resurse
}
```

Construcţia de mai sus putea să conţină şi blocuri `catch`, blocul **`finally`** executându-se totdeauna, indiferent care dintre blocurile `try` sau `catch` s-a executat.

Instrucţiunea `throw`

Instrucţiunea **`throw`** este folosită pentru a declanşa explicit o excepţie; programatorul poate face acest lucru din două motive mai frecvente:

- **nu are cod specific de tratare a unei excepţii, dar doreşte s-o capteze pentru a o notifica intern** (utilizatorii sau obiectele aplicaţiei să fie încunoştinţate într-un anume fel asupra producerii ei), după care o "rearuncă" pentru a fi tratată după modalitatea de tratare existentă în sistem. Spre exemplu, `ceas`, un obiect construit de utilizator este dotat şi cu o proprietate numită `Mesaj`, care stochează informaţii colaterale asociate obiectului; fixarea unei ore inexistente pentru "alarmă" poate fi notificată printr-un mesaj adecvat.

```
catch(Exception exc)
{
        ceas.Mesaj = "Ora neadmisibila";
        throw;
}
```

- **crează o instanţă a unei excepţii existente în sistem sau chiar o excepţie specifică aplicaţiei şi o "aruncă" spre tratare,** pentru a folosi acelaşi mecanism de gestiune a excepţiilor ca şi framework-ul.

```
string msg = "Valoare in afara intervalului admisibil";
ArgumentOutOfRangeException exceptiaMea =
      new ArgumentOutOfRangeException(msg);
throw exceptiaMea;
```

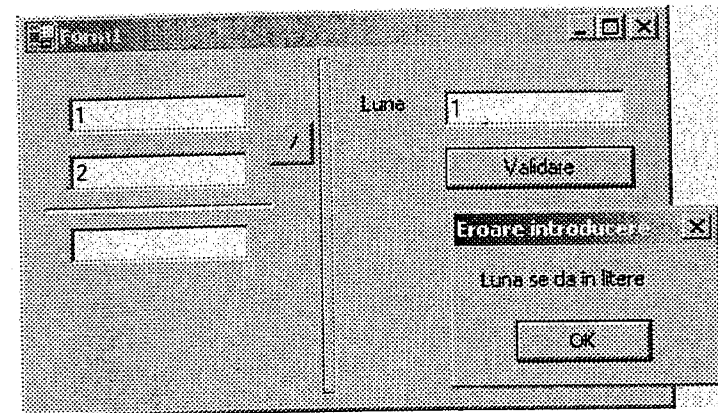
După cum se observă, constructorul clasei aferentă excepţiei are o versiune supraîncărcată, care primeşte valoarea particulară a mesajului de expus la producerea excepţiei respective; această versiune se recomandă ca alternativă la `default constructor`.

Să presupunem acum că dorim să declarăm o excepţie personalizată, specifică aplicaţiei; pentru aceasta, în aplicaţia anterioară, după terminarea descrierii clasei `Form1` se descrie şi clasa aferentă tipului excepţiei dorite. Excepţiile aplicaţiei sunt derivate direct din **`ApplicationException`** şi trebuie să aibă şi default constructor.

```
public class ExceptieDeLuna :
    {
        public ExceptieDeLuna (){ }
        public ExceptieDeLuna (string mesaj) :
                                    base(mesaj)
        {
        }
    }
```


Pe un eveniment (aici *click* pe un buton de validare) se face preluarea valorii de validat, iar la momentul validării se generează sau nu, după caz şi excepţia specifică aplicaţiei.

```
private void Validare_Click(object sender, EventArgs e)
 {
  try
   {
      string luna = DenLuna.Text;
      switch(luna)
      {
       case "ian": case "feb":// ...
                 break;
       default:
            ExceptieDeLuna exc =
                 new ExceptieDeLuna("Luna se da in litere");
            throw exc;
      }
   }
  catch(ExceptieDeLuna exc)
   {
      MessageBox.Show(exc.Message,"Eroare introducere");
   }
 }
```



Desigur mesajul putea fi dat direct în `switch`, dar mecanismul cu excepţii e mult mai general, oferind în plus:

- posibilitatea notificării excepţiei şi altor obiecte;
- coexistenţa cu alte excepţii ce se pot produce în acelaşi bloc **`try`**, tratabile în alte blocuri **`catch`**.

Nu este indicat un apel **throw** din interiorul unui bloc **finally**, deoarece în acel moment există deja încă o excepţie în aşteptarea captării şi tratării.

Atunci când eroarea poate fi tratată pe loc, în blocuri `if-then - else` incluse, nu se recomandă excesul de utilizare a excepţiilor, deoarece fragmentează codul şi face execuţia programului mult mai lentă.

GESTIUNEA MOUSE-ULUI ŞI A TASTATURII

1. **Gestiunea evenimentelor generate de mouse**
2. **Gestiunea evenimentelor generate de tastatură**

1. Gestiunea evenimentelor generate de mouse

Dintre perifericele punctuale, probabil mouse-ul rămâne cel mai utilizat; deşi există combinaţii de taste şi săgeţi pentru toate funcţiile realizate cu mouse-ul, aproape că nu ne putem imagina un program cu o interfaţă grafică evoluată, fără utilizarea acestuia.

Controalele, inclusiv `Form`, interceptează activitatea mouse-ului dacă ele au proprietăţile **Enable** şi **Visible** pe valoarea **True**. Chiar trecerea cu mouse-ul pe deasupra unui control este înregistrată şi poate fi aleasă ca eveniment reper pentru derularea unor acţiuni. Dacă un control `child` este dezactivat, atunci controlul `parent` recepţionează evenimentul legat de mouse. Când mai multe controale sunt suprapuse, numai cel de deasupra primeşte acest tip de eveniment.

În general, se recepţionează un eveniment de mouse doar când mouse-ul se află deasupra controlului, dar acţiuni complexe de genul *drag & drop* pot "captura" mouse-ul, urmărindu-l şi în afara suprafeţei controlului, chiar şi în afara ferestrei aplicaţiei.

Clasa **Control** defineşte nouă evenimente legate de mouse, dintre care patru sunt cele mai frecvent folosite: **MouseDown, MouseUp, MouseMove** şi **MouseWheel**. Lor le corespund funcţiile de tratare moştenite din clasa `Control`: **OnMouseDown, OnMouseUp, OnMouseMove** şi **OnMouseWheel**. Ele dispun şi de un delegat de tip **MouseEventHandler** la care pot fi ataşate şi detaşate (operatorii **+=** şi **-=**) funcţii de tratare a evenimentelor de mouse, scrise de programator.

Funcţia de tratare a unui eveniment de tipul celor de mai sus furnizează şi un parametru de tip **MouseEventArgs** ce conţine informaţii detaliate despre evenimentul produs:

- **X** şi **Y** – coordonatele mouse-ului la momentul producerii evenimentului;
- **MouseButtons** – enumerare de tip `MouseButtons` ce indică butonul (butoanele mouse-ului) utilizat(e);
- **Clicks** – single sau double click (1 sau 2);

- `Delta` – un întreg pozitiv sau negativ ce indică sensul şi dimensiunea deplasării rotiţei mouse-ului.

Pentru a demonstra înlănţuirea evenimentelor de mai sus vom presupune o aplicaţie de "mouse tracking", adică de urmărire a traseului urmat de mouse pe suprafaţa formei.

Exerciţiu

Exemplificaţi **folosirea principalelor evenimente bazate pe mouse** printr-un program care permite extragerea coordonatelor mouse-lui şi desenarea cu mouse-ul. Salvaţi fişierul imagine şi restauraţi-l ulterior pentru a continua desenul.

Scrieţi codul necesar imprimării ulterioare a unui astfel de desen.

Rezolvare

Într-o aplicaţie C# Windows Application dimensionăm forma astfel încât să dispunem de o suprafaţa suficient de mare pentru desen; stabilim `BackColor White`, pentru a fi uşor vizibil desenul. Declarăm o variabilă booleană care arată starea creionului (apăsat, ridicat) şi o listă de puncte, colectate atâta timp cât ţinând apăsat mouse-ul ne deplasăm cu el pe suprafaţa de desen. Eventual o bară de stare va afişa, spre confirmare, starea creionului.

```
ArrayList lstPcte;
private System.Windows.Forms.StatusBar statusBar1;
bool creionApasat;
```

În constructorul formei se iniţializează variabilele declarate anterior.

```
public Form1()
    {
        InitializeComponent();
        creionApasat = false;
        lstPcte = new ArrayList();
    }
```

La **MouseDown** se adaugă un prim punct (cel original) la colecţie, se schimbă starea creionului şi se afişează în bara de stare.

```
private void Form1_MouseDown(object sender,
                System.Windows.Forms.MouseEventArgs e)
```

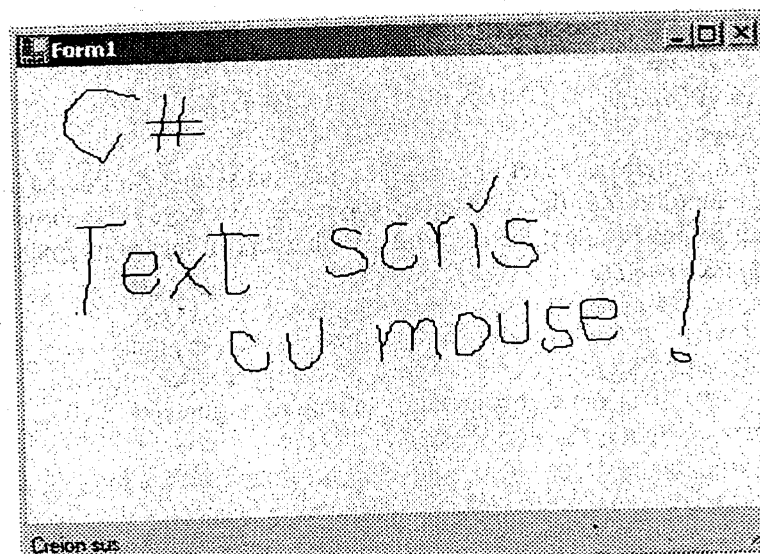


```
{
    Point p=new Point(e.X,e.Y); lstPcte.Add(p);
    creionApasat=true; statusBar1.Text="Creion jos";
}
```

Pe **MouseMove** se preia contextul grafic al formei, se reţine drept punct curent, punctul furnizat de parametrul funcţiei de tratare şi se crează un creion de culoarea preluată din proprietatea `ForeColor` a formei.

```
private void Form1_MouseMove(object sender,
            System.Windows.Forms.MouseEventArgs e)
 {
   Graphics g = this.CreateGraphics();
   Point pctCrt = new Point(e.X,e.Y);
   Pen creion = new Pen(ForeColor);
   if( creionApasat )
   {
    statusBar1.Text="Scrie";
    if(lstPcte.Count > 1)
        g.DrawLine(creion,
            (Point)lstPcte[lstPcte.Count-1],pctCrt);
    lstPcte.Add(pctCrt);
   }
 }
```

Dacă nu este singurul punct din colecţie, se va trasa o linie de la ultimul punct existent în colecţie la punctul curent; punctul curent este apoi adăugat la colecţie, astfel încât cum **MouseMove** se generează repetat pe toată durata deplasării mouse-ului, colecţia să fie completată progresiv.



Pe **MouseUp**, ridicăm creion pentru a putea iniţia un nou şir de puncte desenate şi anunţăm acest lucru în bara de stare.

```
private void Form1_MouseUp(object sender,
                System.Windows.Forms.MouseEventArgs e)
{
  creionApasat = false; statusBar1.Text="Creion sus";
}
```

Un inconvenient major al programului constă în faptul că minimizând fereastra constatăm la maximizare că desenul a dispărut; va trebui rescrisă funcţia `onPaint()` astfel încât să redeseneze toate punctele deja stocate. Probabil că o soluţie eficientă ar fi să definim un masiv bidimensional de puncte, în scară, o linie din matrice preluând toate punctele colecţiei dintre două evenimente **MouseDown** şi **MouseUp**. Am beneficia astfel de facilitatea unei colecţii de a fi extensibilă punct cu punct, dar şi de dimensiunea stabilă a masivelor pentru redesenare, din momentul în care se cunoaşte numărul total de puncte ale unei secvenţe.

Acest lucru uşurează şi serializarea datelor preluând din `ArrayList`, respectiv deserializarea cu retrasarea punctelor, la restaurare dintr-un fişier.

Lăsăm la latitudinea cititorului dezvoltarea programului pentru a răspunde şi celorlalte cerinţe ale exerciţiului.

2. Gestiunea evenimentelor generate de tastatură

Trei sunt evenimentele sesizate în legătură cu folosirea tastaturii:

- **KeyDown**
- **KeyPress**
- **KeyUp**

Ele se generează exact în ordinea de mai sus şi au blocul de parametri ai evenimentului de tipuri diferite. **KeyDown** şi **KeyUp** se produc la apăsarea, respectiv relaxarea unei taste, iar handler-ul lor au argumentul de tipul **KeyEventArgs**.

KeyPress este un eveniment mai special, care se produce la apăsarea unei taste, după **KeyDown** şi doar dacă apăsarea produce o valoare de tip caracter (nu şi la taste funcţionale); blocul de argumente al funcţiei de tratare este de tip **KeyPressEventArgs**.

Evenimentele de tastatură sunt sesizate numai de controalele active; **forma** primeşte şi ea notificarea evenimentelor de tastatură, dar doar dacă a

anunţat prin proprietatea **KeyPreview** pe **true**, că doreşte să intercepteze intrarea de la tastatură, înaintea controlului destinaţie.

Proprietăţi stocate în obiectul KeyPressEventArgs

- **Handled** - precizează prin **true** / **false** dacă evenimentul se consideră deja ca tratat şi nu mai trebuie transmis controlului care are focus-ul în acel moment.
- **KeyChar** – caracterul aferent tastei apăsate şi care a produs evenimentul.

Proprietăţi mai importante stocate în obiectul KeyEventArgs

- **Alt** - `true` dacă tasta **Alt** este apăsată, `false` altminteri;
- **Control** - `true` dacă tasta `Ctrl` este apăsată, `false` altminteri;
- **Handled** – precizează prin `true` / `false` dacă evenimentul se consideră deja ca tratat şi nu mai trebuie transmis controlului care are focus-ul în acel moment;
- **KeyCode** – codul tastei (litere şi taste funcţionale) apăsate; poate fi comparată cu una din valorile enumerării **Keys**.
- **KeyData** – codul tastei apăsate, împreună cu proprietatea **Modifiers** (vezi mai jos);
- **KeyValue** – returnează proprietatea **KeyData** în format **int**;
- **Modifiers** – returnează ce combinaţie de taste au fost apăsate simultan (`Ctrl, Shift, Alt`)
- **Shift** `true` dacă tasta **Shift** este apăsată, `false` altminteri.

Exerciţiu

Să se elaboreze un program care utilizează **principalele evenimente legate de tastatură.** Se va proiecta şi realiza o aplicaţie .NET care simulează un **PIAN,** ce dispune de taste (butoane), dar la care se poate cânta şi **folosind tastatura** calculatorului.

Rezolvare

Aplicaţia **PIAN** exemplifică tratarea unitară, la **nivel de formular** a evenimentelor *keypress* pentru tastatură şi *click* de buton.
Observaţii preliminare:

- un handler poate fi invocat de mai multe evenimente (ex. aceeaşi funcţie tratează şi click de buton şi click pe opţiune meniu) sau mai multe butoane anunţă aceeaşi funcţie de tratare Click.
- un eveniment poate fi tratat prin invocarea mai multor handlere; la rulare poţi detaşa / ataşa dinamic, handlere; ataşarea unui handler pe acelaşi *event* se face cu +=; detaşarea unui handler de la un *event* se face cu -= .

Paşii de realizare:

- Deoarece se va folosi beep-ul de pe procesor, se include un `namespace` specializat pentru accesul la serviciile sistem:

```
using System.Runtime.InteropServices;
```

- La începutul clasei `Form1` se anunţă că aceasta va face un import de funcţie dintr-o bibliotecă legată dinamic, la momentul execuţiei (e vorba de funcţia **`Beep( int nota, int durata )`** folosită la producerea sunetului):

```
[DllImport("kernel32.dll")]
```

- Se tratează evenimentul **`KeyPressed`** la nivel de formă, folosind funcţia următoare:

```
private void Form1_KeyPress
                       (object sender, KeyPressEventArgs e)
{
 int DO = 262,RE = 294,MI = 330, FA = 349, SOL = 392,
                     LA = 440, SI = 494, DO2 = 524;
 switch(e.KeyChar )
        {
              case 'a': Beep(DO,200); break;
              case 's': Beep(RE,200); break;
              case 'd': Beep(MI,200); break;
              case 'f': Beep(FA,200); break;
              case 'g': Beep(SOL,200);break;
              case 'h': Beep(LA,200); break;
              case 'j': Beep(SI,200); break;
              case 'k': Beep(DO2,200);break;
        }
}
```

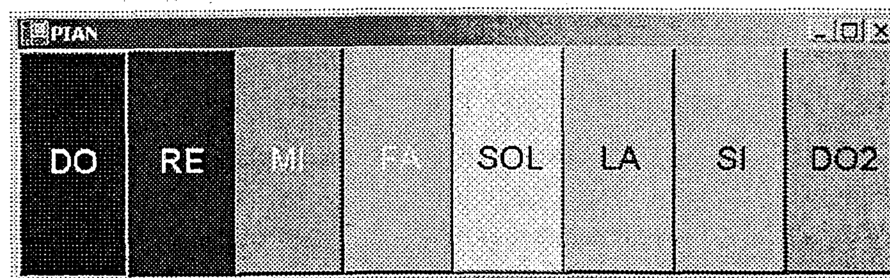
- `Form1` trebuie să aibă setată `true` proprietatea **`KeyPreview`**, care anunţă că forma doreşte să primească mai întâi ea informaţia tastată, pe care o

va direcţiona apoi către TextBox-ul care primeşte focus-ul în acel moment (altfel forma nu primeşte input-ul de la tastatură).

- Se pune o funcţie **`OnButoane()`** la nivel de formă şi ea se declară drept handler pentru evenimentul `click` la nivelul fiecărui buton:

```
private void OnButoane(object sender, System.EventArgs e)
  {
      string nume=((Button)sender).Text;
      int DO = 262,RE = 294,MI = 330, FA = 349, SOL = 392,
                         LA = 440, SI = 494, DO2 = 524;
      switch(nume )
      {
            case "DO": Beep(DO,200); break;
            case "RE": Beep(RE,200);break;
            case "MI": Beep(MI,200);break;
            case "FA": Beep(FA,200);break;
            case "SOL": Beep(SOL,200);break;
            case "LA": Beep(LA,200);break;
            case "SI": Beep(SI,200);break;
            case "DO2": Beep(DO2,200);break;
      }
  }
```

- Se poate specula succesiunea evenimentelor **`KeyDown, KeyPress`** (care se generează repetat dacă se menţine tasta apăsată) şi **`KeyUp`** (generat o singură dată, la eliberare tastă) pentru a modifica durata sunetului.



Pentru a evita repetarea definirii notelor la nivelul ambelor funcţii, acestea se puteau defini la nivelul formei, sau şi mai elegant, ca **o enumerare** :

```
public enum note { DO=262,RE=294,MI = 330, FA = 349,
                   SOL = 392, LA = 440,SI = 494, DO2 = 524};
```

folosită apoi **în ambele funcţii** sub forma:

```
case "DO": Beep((int)note.DO,200); break;
```

Exerciţiu

Să se proiecteze si realizeze sub Visual C# o aplicaţie **calculator**, similară celei din Windows Accessories. Tastele funcţionale vor fi inscripţionate cu roşu, iar cele numerice cu albastru; **introducerea numerelor se poate face de la tastatură sau din butoanele-tastă ale aplicaţiei.** Toate calculele efectuate la o rulare vor fi salvate într-un fişier text, organizat ca pe o bandă de hârtie, putând fi consultate ulterior cu orice editor de text.

Rezolvare

Pentru a nu complica înţelegerea, dăm mai jos doar scheletul de bază al funcţiei de tratare a apăsării pe un buton al calculatorului.

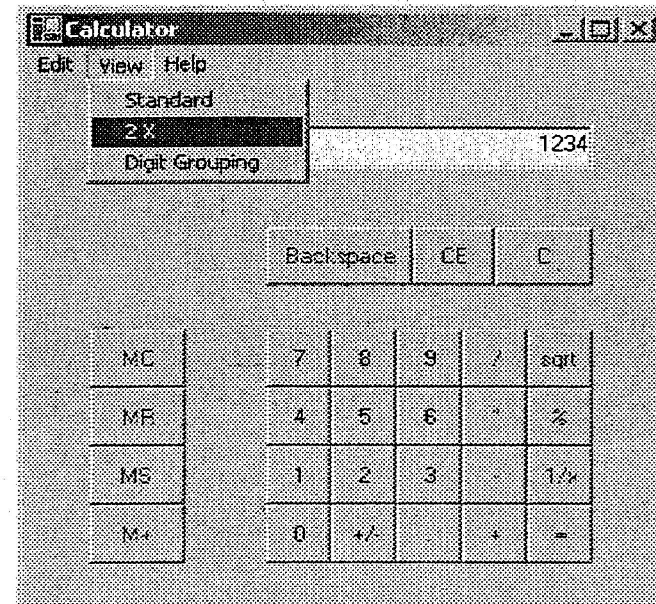
```
private void OnButoane(object sender, System.EventArgs e)
{
      Button b=(Button)sender; string s = b.Text;
      switch(s) // taste functionale
      {
            case "CE": /* tratare Clear Entry */ break;
            case "C" : /* tratare Clear All   */ break;
            // similar pentru alte taste functionale
      }
      if(   char.IsDigit(s[0]) )      //taste cifre
            {
                  // tratare butoane inscriptionate cu cifre
            }

      else // operatori de calcul
      {
            x = double.Parse(tb.Text);
            switch(oldOp)
            {
              case "+":  /* tratare operator + */   break;
                  // similar pentru operatorii - * / ...
              case "=": x=y;     break;
            }
            tb.Text=y.ToString();oldOp=s;init=true;
      }
}
```

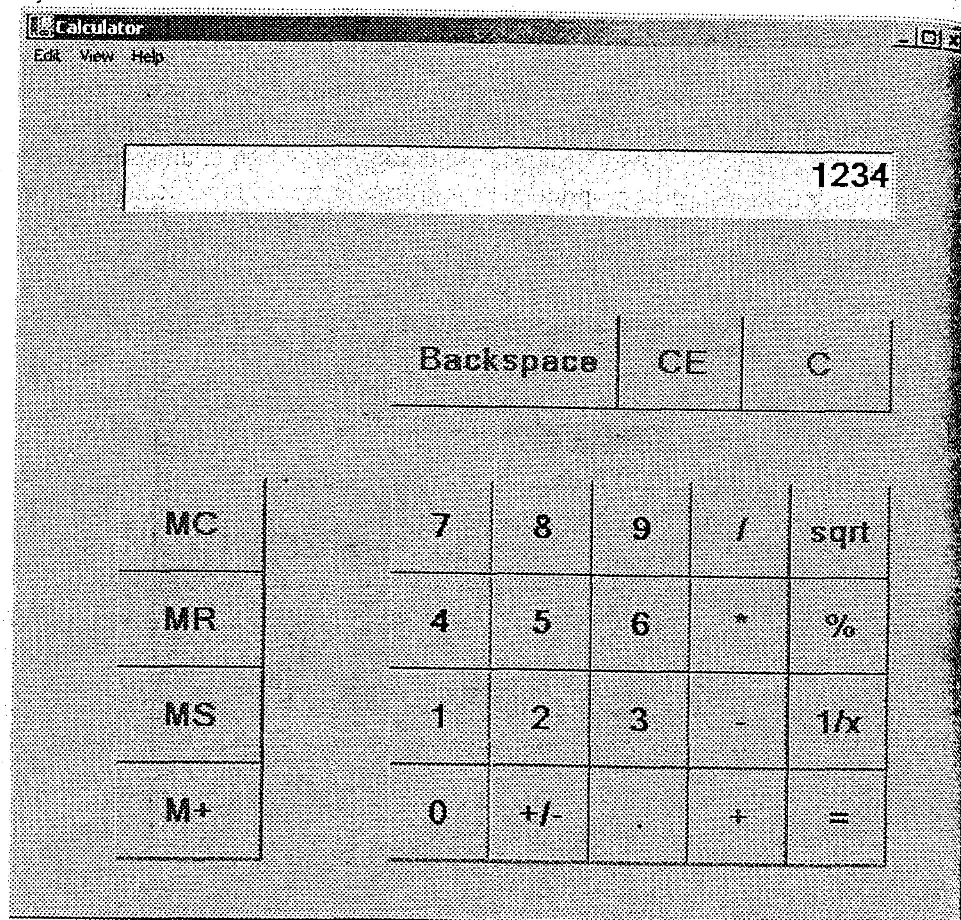

Evident că la apăsarea cifrelor de la tastatură, tratarea ar trebui să fie similară apăsării butoanelor calculatorului; din acest motiv se construieşte un buton temporar care se inscripţionează cu cifra preluată de la tastatură şi se apelează tot funcţia de tratare a apăsării unui buton.

```
private void Form1_KeyPress
                         (object sender, KeyPressEventArgs e)
{
      // proprietatea KeyPreview a formei sa fie pusa pe true
      Button tmp = new Button();
      tmp.Text = e.KeyChar.ToString();
      e.Handled=true;
        // marcheaza ca tratat, sa nu ajunga si la textBox !!
      OnButoane(tmp, new EventArgs());
        // simuleaza apasare de buton
}
```

Uneori este nevoie să redimensionăm forma, pentru a vedea mai bine controalele componente; constatăm că la **Resize**, se măreşte doar fereastra nu şi controalele; aceasta deoarece controalele au poziţii şi dimesiuni fixate în cadrul formei, prin proprietăţile lor şi care ar trebui modificate, una câte una. Funcţia de mai jos, pusă ca tratare a unei opţiuni din meniu, realizează atât dublarea formei, cât şi a controalelor componente.



```
private void menView2X_Click
                    (object sender, System.EventArgs e)
{
    // View 2X dublare imagine cu repozitionare controale
    this.Width*=2; this.Height*=2;
    foreach( Control c in this.Controls)
    {
      // accepta doar modificare la nivel de Location
     c.Location= new Point(c.Location.X*2, c.Location.Y*2);
     c.Height*=2;c.Width*=2;
      // trebuie creat font nou, c.Font fiind imutabil
     Font f=
       new Font(c.Font.Name,2*c.Font.Size,FontStyle.Bold);
     c.Font=f;
    }
}
```



Exerciţiu

Peste o imagine de **telefon mobil** să se plaseze controale care să simuleze funcţiile tastaturii unui mobil: formarea de numere de apel, consultare agendă telefonică, apel de număr, compunere mesaj etc.

Rezolvare

`Opacity` este o proprietate a formei; ea controlează transparenţa formei. Se poate specula proprietatea de opacitate, onorată doar de sistemele de operare capabile să afişeze pe layere (Windows 2000, Windows XP şi ulterioare) pentru a combina culoarea formei cu cea a ferestrelor de sub ea.

`TransparencyKey` este o proprietate a formei, care face transparentă doar o **zonă din fereastră** şi anume cea de culoarea declarată drept culoare de transparenţă. Zonele transparente din fereastră se comportă ca şi cum n-ar fi, adică dând click pe ele mesajul este recepţionat de fereastra de sub fereastra formei !

Nu există o proprietate pentru a gestiona **transparenţa unui control;** se poate totuşi alege pentru proprietatea `BackColor` o culoare creată cu metoda `Color.FromArgb()`, care permite stabilirea unui **factor alpha** de transparenţă. El este detaliat la începutul capitolului de grafică. Spre exemplu, butonului adus din ToolBox, inscripţionat `Exit` şi aşezat peste tasta `Exit` a telefonului i se fixează un factor de transparenţă 25 (din max. 255) şi o culoare albă:

```
Exit.BackColor = Color.FromArgb(25,255,255,255);
```

Dăm în continuare câţiva din paşii mai importanţi pe care trebuie să-i parcurgem pentru a răspunde cerinţelor de mai sus.

- Se descarcă de pe site-ul unei firme de telefonie mobilă imaginea unui telefon mobil, aleasă astfel încât să fie bine vizibile tastatura şi butoanele cu funcţiile de bază ale acestuia.
- Proprietatea `FormBorderStyle` pusă pe valoarea `None`, face ca forma să afişeze imaginea fără border, ca şi cum ar fi vorba nu de o fereastră, ci numai de imaginea telefonului mobil.
- Proprietatea `TransparencyKey` a formei, pusă pe `White`, sau pe culoarea care se consideră ca fundal.

- În constructorul formei, după apelul `InitializeComponent();` se construieşte un obiect **Bitmap BG** pornind de la imaginea telefonului mobil şi se declară ca imagine de fundal pentru formă.

```
Exit.BackColor = Color.FromArgb(25,255,255,255);
Bitmap BG = new Bitmap("nokia.bmp");
this.BackgroundImage = BG;
```

Similar se poate proceda pentru fiecare tastă a telefonului, punând în funcţia de tratare a butonului asociat codul sursă ce simulează acţiunea corespunzătoare. Butonul **Exit** realizează şi ieşirea din aplicaţie.

```
private void Exit_Click(object sender, System.EventArgs e)
        {
                Application.Exit();
        }
```



LUCRUL CU MENIURI ŞI BARE

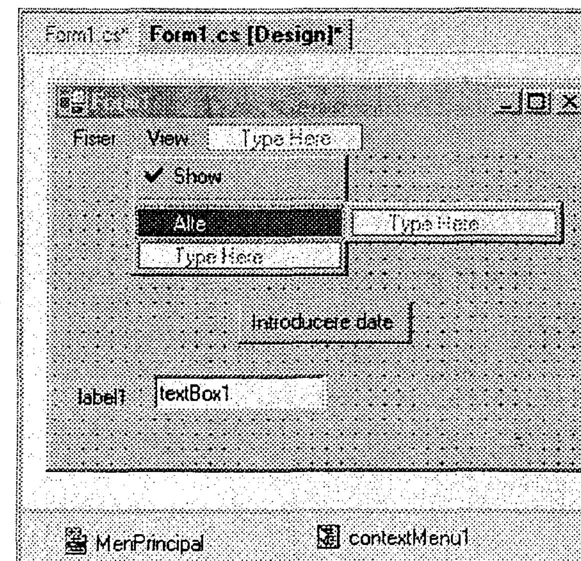
1. **Meniu principal - MainMenu**
2. **Meniuri contextuale - ContextMenu**
3. **Controale de tip bară de instrumente – ToolBar**
4. **Controale de tip bară de stare - StatusBar**

Există două tipuri de meniuri: **principale** (inserabile vizual prin controlul **MainMenu)** şi **contextuale** (inserabile prin controlul **ContextMenu** şi activat la execuţie cu click buton dreapta mouse**).**

Ambele se pot prelua din ToolBox; prin aducerea pe formular vor fi plasate în josul paginii, iar la selectare cu mouse-ul se activează pe platforma aplicaţiei un mecanism *"type here"*, de completare vizuală a opţiunilor.

&x (**x** fiind o literă din denumirea opţiunii precedată de &) acţionează ca accelerator, adică permite selectarea rapidă a opţiunii, cu **Alt + x.**

MainMenu este obiect unic, pe când **ContextMenu** poate avea mai multe instanţe, care se ataşează unor obiecte din form; astfel în funcţie de obiectul pe care dăm click MouseRight (adică în funcţie de context) se va activa unul sau altul dintre meniurile contextuale.



1. Meniu principal - MainMenu

Construirea **vizuală** a unui meniu principal este extrem de simplă:

- Se trage controlul `MainMenu` din ToolBox; el este pus în josul formei; îl redenumim în **Properties / Name** cu numele `MenPrincipal`
- Selectăm meniul şi completăm opţiunile dorite.

Pentru a vedea cum putem face acelaşi lucru **prin program**, fără ajutorul componentei vizuale, este bine să observăm ce a scris Designer-ul pentru noi. În clasa `Form1` apar membrii de tip `MainMenu`, respectiv câte un `MenuItem`, pentru fiecare din opţiunile sale:

```
private System.Windows.Forms.MainMenu MenPrincipal;
private System.Windows.Forms.MenuItem menuItem1;
```

În `InitializeComponent()` sunt instanţiaţi toţi membrii formei, printre care şi cei referitori la meniu, stabilindu-se şi diversele proprietăţi ale acestora:

```
this.MenPrincipal = new System.Windows.Forms.MainMenu();

this.menuItem1 = new System.Windows.Forms.MenuItem();
this.menuItem1.Text = "Fisier";
```

Tot aici sunt incluşi toţi membrii de tip **MenuItem** în colecţia `MenuItems` a meniului principal:

```
this.MenPrincipal.MenuItems.AddRange
      ( new System.Windows.Forms.MenuItem[]
            {
                        this.menuItem1,
                        this.menuItem2
            }
      );
```

Dacă o opţiune se descompune pe subopţiuni, atunci acel item are propria sa colecţie `MenuItems`:

```
this.menuItem2.MenuItems.AddRange
      ( new System.Windows.Forms.MenuItem[]
            {
            this.menuItem3,
            this.menuItem4,
            this.menuItem5
            }
);
```


În timpul completării opţiunilor dintr-un meniu, pe butonul mouse dreapta ni se oferă următoarele posibilităţi:

1. **Insert New** – pentru a adăuga o nouă opţiune;
2. **Delete** – pentru a şterge o opţiune;
3. **Insert Separator** – pentru a introduce un separator între opţiuni;
4. **Edit Names** – care bifat, asigură vizibilitatea şi editarea directă a identificatorilor de opţiuni, nu numai a textului afişat de opţiuni (caption). Denumind sugestiv opţiunile avem avantajul că atunci când designer-ul va completa numele unei funcţii de tratare a opţiunii respective va da un nume funcţiei pornind de la numele opţiunii, mărind astfel lizibilitatea programului.

> Se pot schimba între ele opţiunile deja completate, prin tragere cu mouse-ul.

Meniurile au **proprietăţi** specifice; dintre proprietăţile mai importante amintim:

- `Checked` – precizează prin true / false starea bifat / nebifat pentru opţiunile cu bifă (check mark) sau de tip radio button; pentru un meniu cu opţiuni de tip radio button, excluderea mutuală a opţiunilor ce fac parte din acelaşi grup nu se face implicit, ci prin cod sursă furnizat de programator. Momentul bifării / debifării este de obicei cel legat de evenimentul `Popup`.
- `OwnerDraw` – true / false, dacă programatorul furnizează cod sursă pentru desenarea meniului, în locul celui standard, folosit de Windows.
- `RadioCheck` – cere ca meniul să afişeze un radio button în locul bifei, când proprietatea `Checked` este setată pe `true`.
- `Shortcut` – specifică tastele funcţionale pentru acces rapid la opţiunile unui meniu.

Evenimente specifice

- **Select** – declanşat când la execuţie selectăm opţiunea din meniu (cu mouse sau cu săgeţile de la tastatură);
- **Popup** – când se "desfăşoară" meniul; legat de acest eveniment putem să actualizăm dinamic meniul, conform noului context.
- **Click** – la click de mouse;

- **DrawItem** – generat la desenarea meniului, doar pentru meniurile desenate de programator (proprietatea `OwnerDraw` este `true`), pentru a putea furniza cod de pictare meniu.
- **MeasureItem** – activat când la execuţie se citeşte sau se scrie proprietatea `ItemHeight` (prin accesorii `get-set`).

Handlere-le de tratare a unei opţiuni se pun selectând opţiunea şi dând în **Properties** – **Events** numele funcţiei, apoi completăm în codul sursă ce acţiuni dorim să fie executate la producerea acelui eveniment.

Exerciţiu

Pe o opţiune din meniu `View` ascundeţi / refaceţi butonul de actualizare date, dintr-o aplicaţie de introducere date.

Rezolvare

Se va proceda aşa cum se descrie mai sus despre lucrul cu meniuri, iar pe opţiunea inscripţionată `Show`, de nume `menuItem3`, declarată cu proprietatea `Checked` pe `true`, se adaugă funcţia de tratare a evenimentului `Click`:

```
private void menuItem3_Click
        ( object sender, System.EventArgs e)
{
    button1.Visible = !button1.Visible;
    menuItem3.Checked = !menuItem3.Checked;
}
```

`button1` este şi el un control existent deja pe formă; se observă că nu este nevoie de *if* pentru a comuta dintr-o stare în alta o proprietate cu două valori, `true` şi `false`, ci putem doar să "negăm" starea existentă la un moment dat.

La crearea meniului se poate stabili şi combinaţia de taste ce permite accesul rapid la opţiunea din meniu; în acest sens se va folosi proprietatea **Shortcut**. Combinaţia poate fi apoi folosită, dar afişarea ei pentru informarea utilizatorilor, nu este implicită, ci depinde de proprietatea **ShowShortcut**, care poate fi `true` sau `false`.

Dând *Enter* când este activat mecanismul `TypeHere` de completare a unui meniu, se adaugă un separator; un separator între opţiuni se poate introduce şi folosind MouseRight şi alegând `Insert Separator`.

Se pot muta cu mouse-ul nu numai opţiunile, atunci cînd vrem să le reordonăm, ci putem muta chiar categorii întregi, selectându-le şi deplasându-le cu drag & drop.

2. Meniuri contextuale - ContextMenu

Se trage controlul **ContextMenu** din ToolBox; el este pus în partea de jos a formei ; îl redenumim (cu **Properties / Name**) `contextMenuForma`, pentru că dorim să se activeze pe MouseRight click, dat pe formă.

Meniurile contextuale pot fi **ataşate** câte unui control; asocierea meniului cu un control se face punând în proprietatea `contextMenu` **a controlului** căruia dorim să-i ataşăm meniul, numele meniului contextual dorit. Spre exemplu, în proprietatea `contextMenu` a formei se citează meniul `contextMenuForma`, creat anterior.

Completarea opţiunilor meniului : selectăm meniul contextual din josul formei; în partea superioară a formei apare un mecanism *"type here"*, care permite editarea opţiunilor meniului ca şi la MainMenu; se pot da opţiuni pe verticală şi pe orizontală.

Pentru tratarea evenimentelor pe opţiunile care nu se mai descompun, procedăm ca la meniul principal.

Exerciţiu

Ataşaţi meniuri flotante unor obiecte grafice dintr-o aplicaţie formular:
- pe un meniu contextual al unui TextBox să se introducă posibilitatea de a alege o culoare de scriere şi o culoare de fundal;
- pe un control Label, meniul contextual să dea posibilitatea alegerii culorii, fontului etc.

Rezolvare

1. Se aduce controlul **label1;**
2. se construieşte meniul contextual **conMenLabel**, având ca opţiune `Culoare` şi subopţiuni `Rosu`, `Verde` etc.
3. Se leagă meniul contextual de controlul **label1** cu **Properties / ContextMenu,** selectând din ComboBox-ul afişat de sistem (sau scriind pur şi simplu) numele meniului dorit, `conMenLabel`.
4. Se pune funcţia de tratare pe click subopţiune "Culoare rosie" fie din **Properties / Events** Click, fie dând click de mouse în timpul editării meniului, când avem opţiunea selectată ; apoi se editează codul:

```
label1.BackColor = System.Drawing.Color.Red;
```

Dacă se doreşte ca opţiunea să ofere posibilitatea de a stabili orice culoare din paleta de culori se poate invoca un control specializat:

```
ColorDialog colorDialog1 = new ColorDialog();
if(colorDialog1.ShowDialog() == DialogResult.OK)
{
    label1.BackColor = colorDialog1.Color;
}
```

Acest cod instanţiază un dialog de alegere a unei culori din paleta de culori şi preia culoarea selectată în controlul `colorDialog` drept culoare de fundal pentru controlul `label1`.

5. Similar se pune handler pe click subopţiune "Font Sans Serif " şi se editează codul:

```
label1.Font=
   new System.Drawing.Font
      ( "Microsoft Sans Serif", 16.25F,
         System.Drawing.FontStyle.Bold,
         System.Drawing.GraphicsUnit.Point,
         (System.Byte)(0))
      );
```

Dacă se doreşte ca opţiunea să ofere posibilitatea de a stabili orice font din cele disponibile, poate fi invocat un control specializat:

```
FontDialog fontDialog1 = new FontDialog();
if(fontDialog1.ShowDialog() == DialogResult.OK)
    {
        label1.Font = fontDialog1.Font;
    }
```

Pentru testarea noilor setări, se cere rescrieerea şi textului afişat de etichetă :

```
label1.Text="Nou";
```

Modul de utilizare a meniului **conMenLabel**, prezentat mai sus, are dezavantajul că este legat de un singur control, cel de tip **Label**. În general, un meniu contextual se ataşează simultan mai multor controale; spre exemplu, se doreşte ca nu numai eticheta, ci şi un textbox să beneficieze de acest meniu de stabilire a culorii de fundal şi a fontului de afişare; vom numi în acest caz meniul contextual mai general, **contextMenu1**.

El va fi ataşat ambelor controale (`label1` şi `textBox1`), selectându-le pe amândouă deodată şi inserând la proprietatea **ContextMenu** numele meniului popup, **contextMenu1**.

Din codul sursă de tratare a evenimentului **menuItemClick** va trebui eliminat identificatorul **label1**, deoarece de data aceasta efectul poate viza fie controlul Label, fie controlul TextBox. El trebuie înlocuit cu un control generic şi anume cel aflat sub mouse la momentul activării meniului contextual.

Soluţia o oferă proprietatea SourceControl, care permite identificarea controlului aflat sub mouse când s-a activat opţiunea din meniul contextual:

```
if(contextMenu1.SourceControl == textBox1)
 {
  // contextMenu1.MenuItems.Remove(menuItem4);
  // contextMenu1.MenuItems.Add(menuItem10);
  // MessageBox.Show
  //   ("contextMenu1 activat pentru textBox1");
 contextMenu1.SourceControl.Text= "Activat pentru textBox1";
 }
else if(contextMenu1.SourceControl == label1)
 {
  MessageBox.Show("contextMenu1 activat pentru label1");
  // contextMenu1.MenuItems.Add(menuItem13);
 }
```

După cum se vede din instrucţiunile puse în comentariu, este posibil să adaptăm dinamic opţiunile meniului în funcţie de controlul ţintă, adăugând sau eliminând unele opţiuni. Acest lucru face şi mai flexibilă utilizarea aceluiaşi meniu contextual, cu mici modificări funcţionând pentru şi mai multe controale.

De asemenea, controlul ţintă poate fi identificat univoc printr-o construcţie de forma **contextMenu1.SourceControl** şi lui i se pot adresa direct diverse comenzi, ca în instrucţiunea de mai jos :

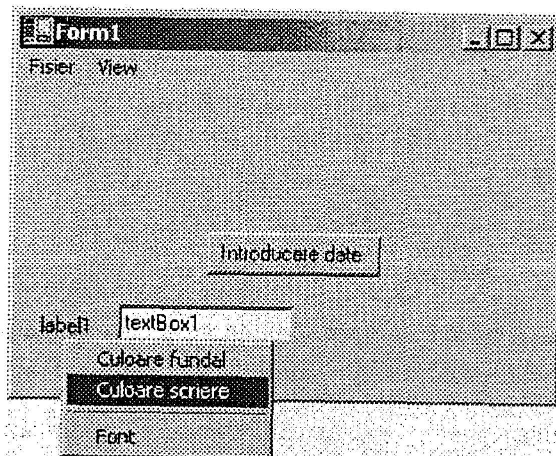
```
contextMenu1.SourceControl.Text = "Activat pentru textBox1";
```

Cu această generalizare, funcţiile de tratare a opţiunilor pentru alegerea culorii de scriere, respectiv a fontului şi dimensiunii acestuia, ar putea arăta astfel:

```
private void menuItem8_Click
                    (object sender, System.EventArgs e)
{
      ColorDialog colorDialog1 =new ColorDialog();
      if(colorDialog1.ShowDialog() == DialogResult.OK)
      {
            contextMenu1.SourceControl.ForeColor =
                        colorDialog1.Color;
```

```
        }
}

private void menuItem7_Click
                (object sender, System.EventArgs e)
{
  FontDialog fontDialog1 = new FontDialog();
      if(fontDialog1.ShowDialog() == DialogResult.OK)
      {
       contextMenu1.SourceControl.Font = fontDialog1.Font;
      }
  contextMenu1.SourceControl.Text="Nou";
}
```



3. Controale de tip bară de instrumente – ToolBar

ToolBar-urile sunt controale care sub forma unor bare constituite din butoane oferă posibilitatea declanşării rapide a unor acţiuni, frecvent folosite într-o aplicaţie.

Lucrul cu ToolBar este relativ simplu:

- se trage din ToolBox un control de tip **ToolBar**; el se plasează automat sub forma unei bare, sub bara de titlu sau sub meniul principal al aplicaţiei (docare `Top`), dar suportă şi celelalte tipuri de docări.
- se aduce din ToolBox un **ImageList** pentru a putea folosi şi butoane cu icon-uri
- selectând `listImage1` plasat sub formă, se denumeşte lista de imagini cu **lst_img** (**Properties / Name**) se populează (**Properties / Images**

Collection + Add) selectând cu browser-ul diverse fişiere de tip *ico*, *bmp* etc.

- cu `toolBar1` selectat, **Properties / Buttons Collection** ( ) + **Add** permite adăugarea de butoane în bara de instrumente; pentru fiecare buton se pot stabili proprietăţi ca:
  - nume, pentru identificare ulterioară
  - stil *(Pushbutton, ToggleButton, Separator, DropDownButton)*
  - textul de afişat
  - indexul imaginii de afişat ca icon, pe buton.

Evenimentele se pot pune numai pe ToolBar în ansamblu, particularizarea acţiunii făcându-se doar prin identificarea butonului care a generat evenimentul; identificarea se poate face după textul afişat pe buton, după numele butonului (butoanele sunt obiecte individuale de tip `System.Windows.Forms.ToolBarButton` recunoscute prin variabile şi la nivelul formei), sau după poziţia pe care o ocupă în cadrul colecţiei de butoane (folosind operatorul de indexare []).

```
private void toolBar1_ButtonClick
            (object sender, ToolBarButtonClickEventArgs e)
 {
      if(e.Button.Text=="but1") // identificare dupa text
       MessageBox.Show
            ("\nS-a apasat butonul inscriptionat but1");
      else
       if(e.Button==b2) // identificare dupa nume
           MessageBox.Show("\nS-a apasat butonul numit b2");
       else
           if(e.Button == toolBar1.Buttons[3])
                 // identificare dupa pozitie in bara
           MessageBox.Show("\nS-a apasat b4 de pe poz 3");
           else
            {
                 MessageBox.Show("\nApasat"+ e.Button.Text);
                 b3.Visible=false;
            }
      int poz = toolBar1.Buttons.IndexOf(e.Button);
            // identificare pozitie
 }
```

De cele mai multe ori, butoanele unui ToolBar se asociază cu aceleaşi acţiuni executate şi când selectăm opţiunile unui meniu. Să ne amintim câte din programele cu care lucrăm permit deschiderea, salvarea,

copy sau *paste*, atât cu opţiuni din meniul *File* sau *Edit*, cât şi din butoanele cu icon-urile uşor de recunoscut ale unui toolbar.

Pentru a nu scrie de două ori acelaşi cod sursă aferent acţiunilor de executat, codul este pus doar în funcţia de tratare a evenimentului Click pe opţiune meniu.

Opţiunea respectivă este apoi declarată în proprietatea **`Tag`** a unui buton din ToolBar, asociind-o astfel butonului cu care are funcţionalitate comparabilă. Spre exemplu, primul buton din toolBar afişează un icon de `Open` şi declanşează aceleaşi acţiuni ca opţiunea **`mnuFisierOpen`** :

```
toolBar1.Buttons[0].Tag = mnuFisierOpen;
```

La click de mouse, butoanele nu tratează evenimentul, ci doar îl redirecţionează către opţiunea de meniu asociată fiecăruia, şi-i cer să-l trateze simulând că s-a executat un click pe opţiune:

```
private void toolBar1_ButtonClick
              (object sender, ToolBarButtonClickEventArgs e)
{
  ToolBarButton tbarButton = e.Button;
  MenuItem mnuItem = (MenuItem) tbarButton.Tag;
  mnuItem.PerformClick(); // metoda a optiunii de meniu
}
```

Dezactivarea opţiunii din meniu (`mnuFisierOpen.Enabled = false;`) inhibă posibilitatea declanşării acţiunilor asociate evenimentului Click pe meniu; dacă se doreşte şi inhibarea click-ului provenind din buton, acest lucru trebuie făcut însă distinct (`toolBar1.Enabled = false;` sau punem cu designer-ul proprietatea pe `false`), pentru că inhibarea opţiunii de meniu nu inhibă automat şi butonul asociat.

Alte observaţii:

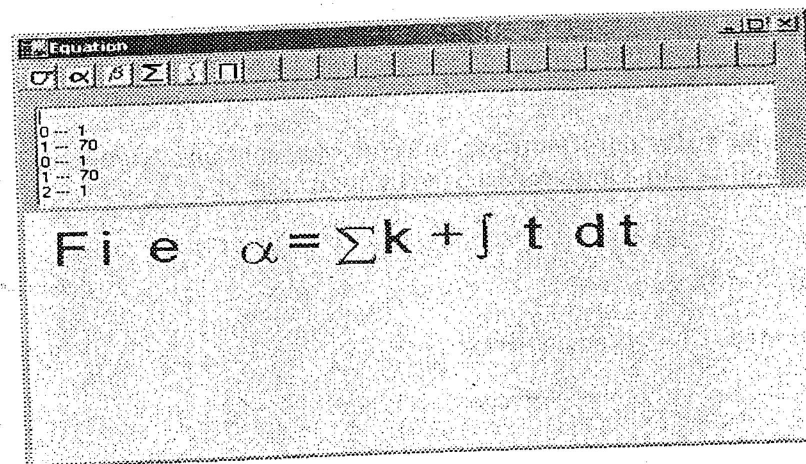
- butoanele pot fi redimensionate global, dacă la nivel de toolBar (nu de buton) declarăm alte dimensiuni pentru butoanele conţinute
- ca la toate controalele, o parte din proprietăţi pot fi schimbate dinamic

Exerciţiu

Să se construiască un **mini-editor grafic cu simboluri** care scrie pe un panel, text introdus de la tastatură sau simboluri grafice asociate unor butoane din toolbar (ca în Word Equation).

Rezolvare

1. Inserare **Panel** (folosit ca zonă pentru desen) cu proprietăţile:
   - `DrawGrid - false, Dock Bottom`
   - `Font Arial, size 30; BackColor Red;`
2. inserare `ToolBar`; adăugăm mai multe butoane mici în colecţia `Buttons` a ToolBar-ului;
3. creare bmp-uri ce vor fi afişate pe butoanele ToolBar-ului:
   - `New / File / bmp`
   - `View / ToolBars / ImageEditor`, dacă nu vedem instrumentele pentru desen în bara de stare a mediului integrat
   - prelucrăm şi salvăm imaginile, de fiecare dată sub alt nume: `alfa.bmp`, `beta.bmp`, `integr.bmp` etc., sugerând simbolurile desenate
   - preferabil să punem un fond gri la desenarea imaginii, pentru asortare cu restul toolbar-ului;
4. un obiect de tip **`listImage1`** este adus din ToolBox şi populat cu bmp-uri (`Properties` / colecţia **`Images`**).
5. Pe `toolBar1` pus **Properties** în `ListImage` se alege **`listImage1`**, creată anterior;
6. asociem imagini fiecărui buton din `toolBar1`: **Properties** / **Buttons** şi **`Add`** pentru fiecare, punând indexul imaginii corespunzătoare din **`listImage1;`**



7. în **Word, Insert / Symbol** alegem font `Symbol` şi căutăm **codurile** afişate pentru simbolurile dorite.
8. Declarare **variabile de lucru**, ca membri în formă:

```
public string txt;          // textul de desenat
public PointF pnt;          // poziţia de scriere
public Font []vfnt;         // fonturi utilizate
```

9. **Iniţializare membri în constructorul formei:**

```
txt=""; pnt =new Point(10,10);
vfnt= new Font[2];
   vfnt[0]= new Font("Arial",30);
   vfnt[1]= new Font("Symbol",30);
```

10. **Tratare Click pe ToolBar**

```
private void toolBar1_ButtonClick( object sender,
                          ToolBarButtonClickEventArgs e)
    {
        int i = toolBar1.Buttons.IndexOf(e.Button);
        txt+=(char)2; // va comuta font in simbol
        char c=' ';
        switch(i)
            {
                case 0: c='s'; break;
                case 1: c='a'; break;
                case 2: c='b'; break;
                case 3: c=(char)229; break;
                case 4: c=(char)242; break;
                case 5: c=(char)213; break;
            }
        txt+=c;
        panel1.Invalidate();
    }
```

11. **Funcţia de redesenare panel, la introducerea câte unui simbol din buton toolbar sau a unui caracter de la tastatură:**

```
private void panel1_Paint(object sender, PaintEventArgs e)
 {
     pnt.X=10; pnt.Y=10;
     Font fnt = vfnt[1];
     for(int i=0;i<txt.Length; i++)
     {
      //textBox1.Text+="\r\n"+i.ToString()+" ---  "
      //   +((int)txt[i]).ToString();
      switch((int)txt[i])
       {
          case 1: fnt=vfnt[0];  break;
          case 2: fnt=vfnt[1];  break;
          default:
```

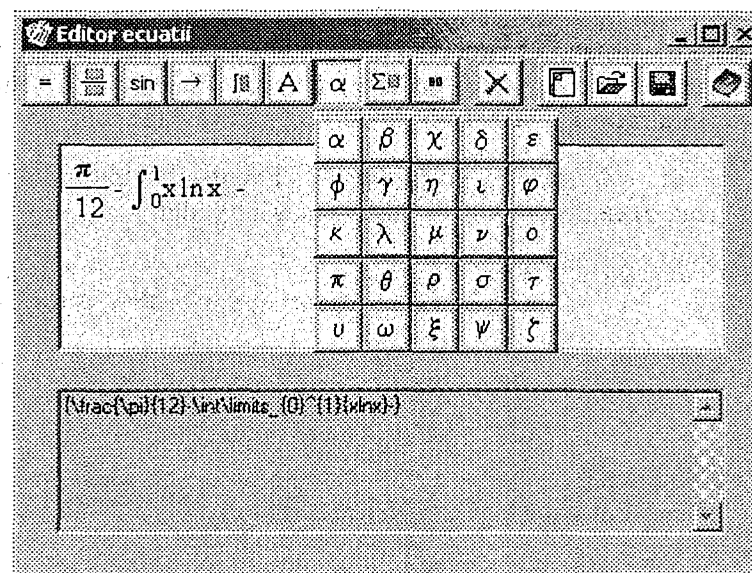


```
                {
                string tmp=""+ txt[i];
                e.Graphics.DrawString
                     ( tmp, fnt, new SolidBrush(Color.Red),
                                                       pnt.X,10   );
                pnt.X+=fnt.Size; break;
                }
           }
       }
 }
```

12. **Tratare introducere text de la tastatură**

```
private void Form1_KeyPress(object sender,
                               KeyPressEventArgs e )
{
     txt+=(char)1; txt+= e.KeyChar;
     panel1.Invalidate();    e.Handled=true;
}
```

Punând proprietatea **Dock** pe **None**, putem apoi redimensiona Toolbar-ul astfel încât să afişeze butoanele pe mai multe rânduri, îl putem face vizibil numai când apăsăm un buton din alt toolbar "master", îl putem plasa peste alt toolbar invizibil, astfel încât să devină vizibil în poziţia în care să ne sugereze că este o dezvoltare a unui buton; spre exemplu, în figura de mai jos plasarea şi vizibilitatea toolbar-ului aferent simbolurilor greceşti sunt corelate cu butonul corespunzător din Toolbar-ul central de simboluri.



4. Controale de tip bară de stare - StatusBar

Prezentăm în continuare câteva elemente de bază ce ţin de lucru cu o bară de stare.

- se aduce din ToolBox controlul **StatusBar**; el se numeşte `statusBar1` şi se plasează implicit pe formular, în partea de jos;
- cu `statusBar1` selectat, **Properties / Panels** permite adăugarea de elemente la colecţia `Panels` deţinută de bara de stare; ne asigurăm că ele sunt şi vizibile (`statusBar1` are proprietatea `ShowPanels` pe `true`); vom adăuga **patru** astfel de panele.

Afişarea orei exacte în ultimul panel din StatusBar

- pe formă, se aduce din ToolBox un control de tip `Timer`, specializat în notificarea scurgerii unui interval de timp; el e plasat automat sub formular, nu va avea parte vizibilă la rulare, dar forma îl recunoaşte, iar prin program putem beneficia de serviciile lui.
- cu `timer1` selectat, în **Properties** fixăm proprietatea `Interval` pe valoarea 1000 (adică la o secundă, deoarece intervalul de timp dorit se exprimă în milisec.)
- cu `timer1` selectat, în **Properties** fixăm `Enabled` pe `true`, altfel (adică rămânând pe `false`, deci inhibat) obiectul Timer nu va semnala evenimentul `Tick`, fiind candidat la ştergere prin garbage collector.
- cu `timer1` selectat, în **Properties** + **Events** se tratează unicul eveniment, `Tick`, recunoscut de timer, adăugând codul:

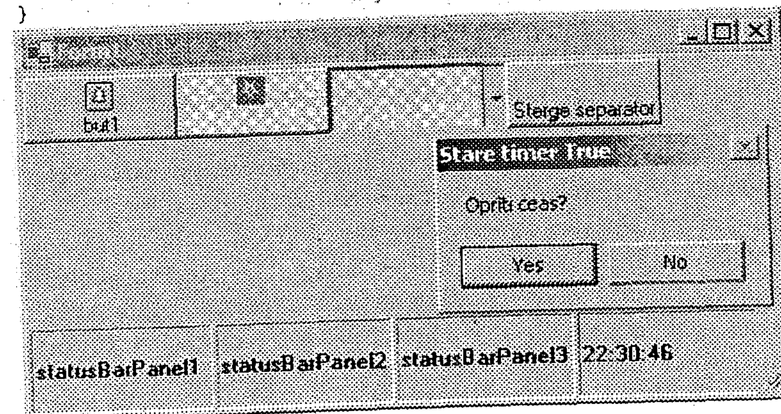
```
private void timer1_Tick
                    (object sender, System.EventArgs e)
{
   statusBar1.Panels[3].Text =
            DateTime.Now.ToLongTimeString();
}
```

ceea ce înseamnă că la fiecare secundă se modifică textul înscris în ultimul panel din statusBar, prin afişarea orei exacte.

Sesizarea panelului pe care s-a dat click de mouse

StatusBar recunoaşte printre evenimente şi `PanelClick`; cu `statusBar1` selectat, **Properties + Events** oferă posibilitatea vizualizării evenimentelor recunoscute de acesta; dând click pe `PanelClick` sau completând un nume de funcţie şi dând Enter, putem adăuga cod sursă pentru acţiuni individualizabile la nivel de panel; pentru varietate, **sesizarea panelului pe care s-a dat click** se va face folosind o metodă aparţinând colecţiilor (`IndexOf`) care primeşte ca parametru panelul care a generat evenimentul şi-l identifică, returnându-ne poziţia pe care acesta o are în colecţie.

```
private void statusBar1_PanelClick(object sender,
                    StatusBarPanelClickEventArgs e)
{
 int i = statusBar1.Panels.IndexOf(e.StatusBarPanel);
 switch (i)
      {
      case 3 :
            bool activat = timer1.Enabled;
            bool stop = MessageBox.Show("Opriti ceas?",
            "Stare timer "+ activat, MessageBoxButtons.YesNo)
                         == DialogResult.Yes;
            if( stop && activat )
                  {  timer1.Enabled=false;
                     statusBar1.Panels[3].Text=" ";  }
                  break;
      case 1 :
            MessageBox.Show("Ati apasat Panel 2."); break;
      default :
            MessageBox.Show("Ati apasat Panel "+(i+1));break;
      }
}
```



CONTROALE COMPLEXE DE VIZUALIZARE

1. **Vizualizare liniară - controlul ListView**
2. **Vizualizare arborescentă – controlul TreeView**

1. Vizualizare liniară - controlul ListView

Există câteva controale numite şi **controale de vizualizare**, deoarece ele oferă suportul pentru vizualizarea informaţiilor sub diverse formate: liniare, arborescente, rapoarte, liste de iconuri etc. Prin formatele sale de vizualizare, `Details`, `LargeIcon`, `SmallIcons` sau `List`, controlul **ListView** este unul dintre cele mai des folosite.

Mecanismul de lucru cu ListView

1. Folosind Visual .NET se generează scheletul unei aplicaţii C# de tip Windows Forms.
2. Se aduce prin drag & drop controlul **`ListView`** din ToolBox; pentru simplitate, denumim (cu **Properties / Name**) acest control `lv`.
3. Pentru că o listă de vizualizare foloseşte şi iconuri pentru reprezentarea simplificată a informaţiei, avem nevoie şi de un control **`ImageList`** adus tot din ToolBox; mediul îl plasează pe linia de jos a machetei aplicaţiei; denumim această listă de imagini cu `il`.
4. Populare listei cu imagini: cu obiectul `il` selectat, folosind **Properties / Images Collections** "..." `Add` adăugăm imagini preluate din fişiere ***.bmp**. În capitolul despre controlul ToolBar se dau mai multe detalii privind crearea şi editarea imaginilor cu editorul specific mediului Visual Studio, dar aceste imagini pot fi construite şi cu editoare externe.
5. Obiectul listă de imagini construit mai sus **trebuie asociat cu controlul de vizualizare;** în acest scop, **ListView** în **Properties** conţine şi proprietăţile **`Large Image List`**, respectiv **`Small Image List`**; ele vor fi legate de lista creată mai sus, alegând din combo lista de imagini **`il`**.
6. Proprietatea **`View`** a obiectului `lv` stabileşte formatul de vizualizare al listei (**`Details`**, **`LargeIcon`**, **`SmallIcons`** sau **`List`**).
7. **Configurare iniţială a coloanelor (capul de tabel)**
   - **lv / Properties / Columns / Collections (...) + Add**; apelăm `Add` repetat, pentru câte coloane adăugăm în listă; pentru fiecare membru adăugat stabilim totodată şi textul de afişat (proprietatea `Text`) ; numele coloanei *(column header)* putem să-l lăsăm cum îl generează automat sistemul sau putem să-l modificăm pentru a-l face mai sugestiv.
   - **Observaţie**. Pentru a putea să şi vedem coloanele pe măsura configurării lor statice, **lv** trebuie să aibă formatul de vizualizare (proprietatea **`View`**) pus în prealabil pe **Details**.
8. **Popularea statică** (la momentul proiectării aplicaţiei) a listei de vizualizare:
   - **lv / Properties / Items / Collections (...) + Add**; apelăm `Add` repetat, pentru câte linii adăugăm; pentru fiecare linie adaugată stabilim totodată şi textul de afişat în prima coloană a liniei (proprietatea **`Text`**) când formatul de vizualizare este `Details`; el este cel care se afişează şi în formatul **`List`** şi sub imagini, când formatul de vizualizare este **`LargeIcons`** sau **`SmallIcons`**.
   - **lv / Properties / Items / Collections (...)** ; pentru fiecare item pus anterior, în partea dreapta proprietatea **`Data`** are **`SubItems`** / **`Collections`** (...) care deschide o fereastra similara cu cea de la `Items`; cu **`Add`** punem câte subitem-uri dorim; subitem înseamnă coloanele fiecărei linii; o matrice este văzută deci ca o colecţie de linii **(items)**, iar fiecare linie este o colecţie de coloane **(subitems)**.
   - **Observaţie:** primul subitem e pus automat şi coincide cu item; orice modificare îl modifică si pe cel din item; deci în mod obişnuit trebuie lăsat aşa.
9. **Populare dinamică** (la momentul execuţiei) se poate face preluând datele dintr-o structură de date:

```
struct date
{
      public int Marca;
      public string Nume;
      public double Salariu;
      public date(int m, string nm,double s)
            { Marca=m; Nume=nm; Salariu=s; }
}
private void btnCompletari_Click
                        (object sender, System.EventArgs e)
{
      lv.Columns.Add("Obs",20,HorizontalAlignment.Right);
      date []vs; vs=new date[2];
      vs[0]=new date(400,"Adamescu Ion", 5500000);
      vs[1]=new date(500,"Banica Paul", 6500000);
      lv.Items.Add("300");
      foreach(date x in vs)
      {
```

```
        ListViewItem itm =
                new ListViewItem(x.Marca.ToString(),1);
        itm.SubItems.Add(x.Nume);
        itm.SubItems.Add(x.Salariu.ToString());
        lv.Items.Add(itm);
        }
  }
```

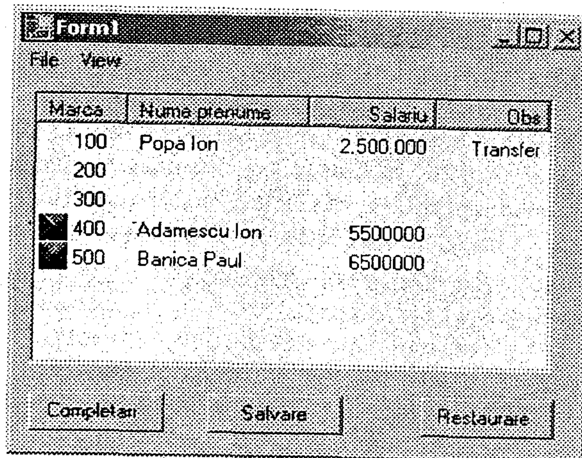
10. Pe meniu **View** putem schimba formatul de vizualizare:

```
private void menDetaliu_Click
                        (object sender, System.EventArgs e)
 {
        lv.View = View.Details;
 }

private void menSmallIcon_Click
                        (object sender, System.EventArgs e)
 {
     lv.View = View.SmallIcon;
 }
```

11. La schimbare item selectat, putem **schimba culoarea de scriere**:

```
private void lv_SelectedIndexChanged
                (object sender, System.EventArgs e)
{
  foreach( ListViewItem itm in lv.Items)
   {
     if(itm.Selected)itm.ForeColor = System.Drawing.Color.Red;
      else  itm.ForeColor = System.Drawing.Color.Blue;
   }
}
```



12. Tratarea evenimentului `DoubleClick`:

```
private void lv_DoubleClick
                        (object sender, System.EventArgs e)
 {
     MessageBox.Show("lv are "+lv.Items.Count+" linii");
 }
```

13. Dacă obiectul `lv` are proprietatea `MultiSelect` pe `true`, ni se va permite selectarea simultană a mai multor item-uri. Controlul ţine şi o colecţie cu toate item-urile selectate la un moment dat, evenimentul `ItemActivate` declanşându-se la momentul indicat prin proprietatea `Activation`:

- `ItemActivation.Standard`,
- `ItemActivation.OneClick` sau
- `ItemActivation.TwoClick;`

Putem specula acest eveniment ataşându-i un handler care prelucrează unul sau toate item-urile colecţiei :

```
private void lv_ItemActivate(object sender, EventArgs e)
{
   MessageBox.Show
            ("Activare: " + lv.SelectedItems[0].Text);
}
```

Activarea `ItemActivation.TwoClick;` diferă de cea standard pe `DoubleClick`, pentru că cele două click-uri se pot da acum la orice interval de timp unul după altul, situaţie marcată şi prin schimbarea cursorului (devine de tip "hand"). Formele de activare `OneClick` şi `TwoClick` dezactivează şi proprietatea `LabelEdit`, care va trebui cerută explicit acum.

14. Salvarea item-urilor prin serializare într-un fişier:

```
private void btnSalv_Click
                        (object sender, System.EventArgs e)
{
  FileStream s = new FileStream("pers.dat",FileMode.Create);
  ArrayList lista = new ArrayList();
  foreach(ListViewItem itm in lv.Items) lista.Add(itm);

  BinaryFormatter f = new BinaryFormatter();
  f.Serialize(s, lista);
  s.Close();
}
```

Deoarece colecția **lv.Items nu este serializabilă**, declarăm o altă colecție **ArrayList lista**, pe care după inițializare o încărcăm cu item-urile din vizualizare. Un formater binar preia sarcina serializării noii colecții. Se poate observa că în fișier sunt salvate obiecte complexe, cu toate informațiile despre item (icon, stare etc.)

15. Restaurarea listei prin deserializare din fișier

```
private void btnRest_Click(object sender, System.EventArgs e)
{
  FileStream s = new FileStream("pers.dat",FileMode.Open);
  BinaryFormatter f = new BinaryFormatter();
  ArrayList lista = (ArrayList)f.Deserialize(s);
  s.Close();
  for(int i=0; i< lista.Count; i++)
    {
          ListViewItem itm;
          itm= (ListViewItem)lista[i];
          lv.Items.Add(itm);
    }
 }
```

Abordarea programatică a controlului ListView

Aplicația de mai sus, construită în mare parte vizual, putea fi realizată și scriind doar cod sursă C#. Iată cum ar arăta aceeași aplicație scriind o funcție care instanțiază obiectul ListView, îl configurează și-l populează cu aceleași categorii de informații. În plus, se construiesc și populează **tot prin program** și cele două liste de imagini asociate item-urilor din lista de vizualizare.

Listei de vizualizare i se stabilesc și câteva proprietăți mai deosebite:

- să permită modificarea conținutului item-ului chiar în listă (**LabelEdit**);
- să afișeze item-urile cu un checkbox și sortate crescător, indiferent de ordinea în care le adăugăm noi;
- selecția unui item să producă selecția întregului rând din listă;
- conținutul listei să fie afișat într-un grid;
- coloanele să poată fi reordonate cu *drag and drop*.

```
private void CreazaListView()
{
 ListView lv = new ListView();
 lv.Bounds =
     new Rectangle(new Point(10,10), new Size(300,100));
```



```
lv.View = View.Details; // Vizualizare in format DETAILS
lv.LabelEdit = true; // Editare direct in listView


lv.AllowColumnReorder = true; // Aranjare coloane cu drag&d

lv.CheckBoxes = true; // Atasare check box
lv.FullRowSelect = true; // Select tot, nu doar primul item
lv.GridLines = true; // Grid de separare lin / col
lv.Sorting = SortOrder.Ascending; // Tine items sortate asc

// Creare cap de lista
lv.Columns.Add("Marca", -2, HorizontalAlignment.Left);
lv.Columns.Add("Nume prenume",-2, HorizontalAlignment.Left);
lv.Columns.Add("Salariu", -2, HorizontalAlignment.Right);
lv.Columns.Add("Observatii",-2, HorizontalAlignment.Center);

// Alocare şi încărcare trei item-uri cu persoane
ListViewItem item1 = new ListViewItem("300",0);
          item1.Checked = true;
          item1.SubItems.Add("Popescu Ion");
          item1.SubItems.Add("850");
          item1.SubItems.Add("Transferat");
ListViewItem item2 = new ListViewItem("200",1);
          item2.SubItems.Add("Ionescu Vasile ");
          item2.SubItems.Add("950");
          item2.SubItems.Add("");
ListViewItem item3 = new ListViewItem("100",0);
          item3.Checked = true;
          item3.SubItems.Add("Adamesteanu Ion");
          item3.SubItems.Add("1250");
          item3.SubItems.Add("");

// Populare lista prin adaugarea item-urilor la ListView
lv.Items.AddRange(new  ListViewItem[]{item1,item2,item3});

// Creare liste de imagini asociate.
ImageList imagMici = new ImageList();
ImageList imagMari = new ImageList();

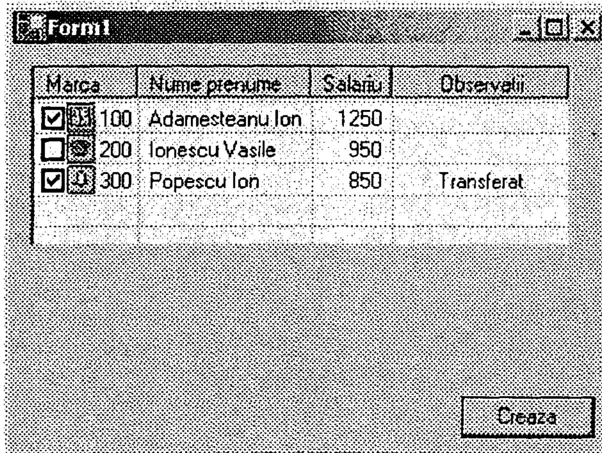
// Populare liste cu imagini din fisiere
imagMici.Images.Add(Bitmap.FromFile("C:\\BELL.bmp"));
imagMici.Images.Add(Bitmap.FromFile("C:\\PHONE.bmp"));
imagMici.Images.Add(Bitmap.FromFile("C:\\CALENDAR.bmp"));

imagMari.Images.Add(Bitmap.FromFile("C:\\HAND.bmp"));
imagMari.Images.Add(Bitmap.FromFile("C:\\ FISH.bmp"));

// Leaga listele cu imagini de listView
```

```
  lv.LargeImageList = imagMari;
  lv.SmallImageList = imagMici;

  // Adauga ListView la colectia de controale a formei
  this.Controls.Add(lv);
}
```



Funcția de mai sus poate fi pusă ca funcție de tratare a unei opțiuni de meniu sau a evenimentului click de buton.

```
private void btnCreaza (object sender, System.EventArgs e)
{
    . CreazaListView();
}
```

2. Vizualizare arborescentă – controlul TreeView

Controlul **`TreeView`** este folosit pentru a afişa o colecție de informații în formă arborescentă, cu posibilitatea expandării sau comprimării unor nivele din arbore.

Fiecare nod al controlului **`TreeView`** este un obiect de tip **`TreeNode`**. Fiecare obiect **`TreeNode`** stochează în proprietatea **`Nodes`** o colecție (**`TreeNodeCollection`**) de subnoduri corespunzătoare unui nivel al arborelui. Proprietatea **`Nodes`** a controlului **`TreeView`** păstrează colecția de noduri aflată la primul nivel (nivelul 0) în arbore. Dacă există noduri pe nivelul 1 al arborelui, acestea vor fi stocate folosind proprietățile **`Nodes`** ale nodurilor situate pe nivelul 0, ş.a.m.d.

Principalele **proprietăți** ale controlului `TreeView`:

- **`Nodes`** (get) – conține colecția de obiecte `TreeNode` asociată controlului sau asociată unui nod de pe nivelurile inferioare;
- **`PathSeparator`** (set / get) – conține delimitatorul folosit pentru calea nodurilor arborelui;
- **`LabelEdit`** (set / get) – indică dacă etichetele nodurilor pot fi editate sau nu;
- **`SelectedNode`** (set / get) – specifică nodul curent selectat al arborelui;
- **`Sorted`** (set / get) – indică dacă nodurile sunt sau nu ordonate alfabetic;
- **`CheckBoxes`** (set / get) – stabileşte dacă fiecare nod al controlului va fi precedat sau nu de câte un *check box*;
- **`ImageList`** (set / get) – precizează controlul de tip `ImageList` care conține icon-urile asociate nodurilor.

Principalele **evenimente**:

- **`AfterSelect`** – este evenimentul implicit al clasei `TreeView`; se declanşează imediat după selectarea unui nod;
- **`AfterLabelEdit`** – este semnalat imediat după editarea etichetei unui nod;
- **`AfterExpand`** – apare imediat după expandarea unui nod.

Mecanismul de lucru cu TreeView

1. În Visual Studio se alege o aplicație C#, tip Windows Application
2. Aducem un control **TreeView** din **ToolBox** şi îl denumim **tv**
3. **Adăugare de noduri statice** se poate face cu:
   - **Properties:** colecția **`Nodes`** şi activez "...";
   - buton **`AddRoot`** şi stabilim textul de afişat în nodul rădăcină;
   - buton **`AddChild`** dacă dezvoltăm un nod existent.
4. **Adăugarea de noduri la momentul execuției** se poate face după modelul următor:
   - **în formă** se defineşte o referință de nod, **`nodSelect`**, pentru a ține nodul curent selectat la un moment dat, de care vom lega noul nod:
     `TreeNode nodSelect;`
   - pe evenimentul **`AfterSelect`** se încarcă referința pe măsură ce ne deplasăm prin arbore, preluând nodul din parametrul funcției de tratare:

```
nodSelect = e.Node;
```

- pe opțiune meniu sau pe alt eveniment, se adaugă la colecția **Nodes** a nodului selectat, un nod nou, furnizând minimul de informație și anume eticheta nodului (preluând-o eventual dintr-un TextBox) după care se cere expandarea nodului curent, sau a întregului arbore:

```
nodSelect.Nodes.Add("Nod Nou");
nodSelect.Expand();
```

Cele două **operații** principalele, **inserare și stergere**, pot fi realizate cu funcții de genul celor care urmează:

- **Inserarea unui nod rădăcină** în controlul **treeView1**, existent la nivelul formei

```
public Form1()
    {
        InitializeComponent();
        //se creaza un obiect de tip TreeNode cu eticheta
        //"nodul 1", nod care se va adauga la controlul
        //TreeView ca nod radacina
        TreeNode nodNou= new TreeNode("nodul radacina");
        // se adauga nodul creat la controlul TreeView,
        treeView1.Nodes.Add(nodNou);
    }
```

- inserarea unui nod copil la nodul curent selectat; dacă nu există nici un nod selectat, noul nod se adaugă tot la nivelul radacină:

```
private void Insereaza(object sender, EventArgs e)
 {
  // se preia nodul selectat
  TreeNode nodCrt=treeView1.SelectedNode;
  // se preia numarul nodurilor controlului Tree View
  int nr = treeView1.GetNodeCount(true)+1;
  // se creaza un nod nou cu eticheta "nodul nn"
  TreeNode nodNou = new TreeNode("nodul "+nr);
  // daca nu exista nici un nod selectat
  if (nodCrt == null)
      // se adauga noul nod lângă celelalte noduri radacina
      treeView1.Nodes.Add(nodNou);
  else
      // noul nod se adauga la colectia nodului selectat
      nodcrt.Nodes.Add(nodNou);
  treeView1.ExpandAll();
 }
```


- **Stergerea unui nod** din controlul TreeView și a tuturor nodurilor copil ale acestuia:

```
private void Stergere(object sender, System.EventArgs e)
{

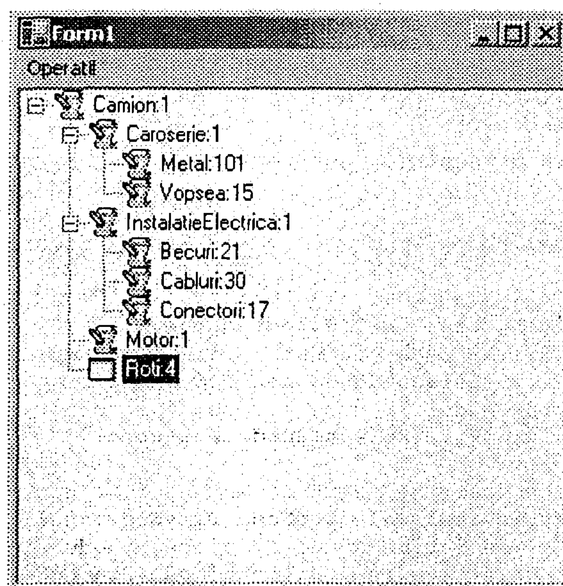
 if(treeView1.Nodes.Count>0) //daca exista noduri in arbore
 {
  //se preia în tn nodul selectat
  TreeNode tn = treeView1.SelectedNode;
  //daca nodul selectat nu este ultimul se retin legaturile
  if (treeView1.SelectedNode.NextNode!=null)
   //se retine nodul ce succeede nodului curent
      tn = treeView.SelectedNode.NextNode;
  else
      // daca nodul selectat nu este primul nod al arb.
       if (treeView1.SelectedNode.PrevNode!=null)
            //se retine nodul ce precede nodului curent
            tn = treeView1.SelectedNode.PrevNode;
  //se sterge nodul selectat
  treeView1.SelectedNode.Remove();
  // nodul retinut in tn devine noul nod selectat
  treeView1.SelectedNode = tn;
 }
}
```

Proprietatea **Dock** se alege **Fill** dacă dorim să umple întreaga formă. Atribuirea de imagini fiecărui nod se face după modelul lucrului cu liste de imagini.

Exercițiu

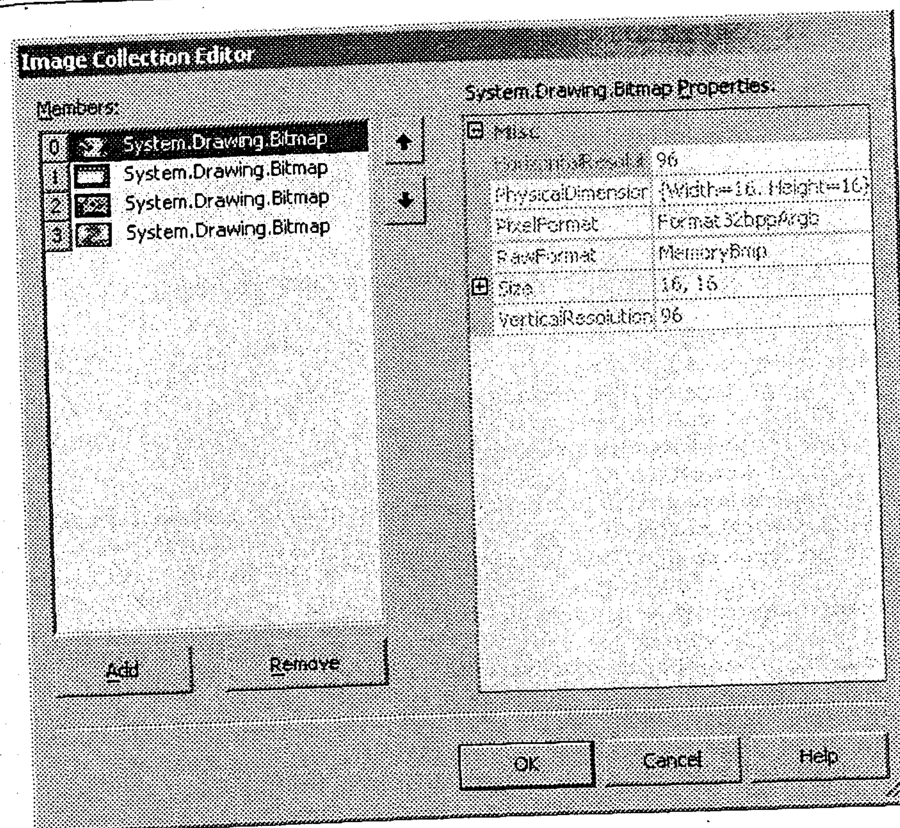
Să se construiască o aplicație care să permită manipularea informațiilor despre structura unor produse; pentru **fiecare produs** se cunosc denumirea și cantitatea produsă, despre **fiecare reper** care intră în componența produsului, denumirea și nr.de repere/produs, iar pentru **fiecare materie primă** utilizată pentru realizarea reperului, denumirea și consumul specific.

Toate perechile de informații sunt prezentate în formatul **denumire:cantitate**. Construirea arborelui se va face dinamic, prin inserarea de noduri. Inserarea, respectiv ștergerea nodurilor se realizeză folosind un meniu contextual. Editarea etichetelor nodurilor se face dând click pe aceasta.



Rezolvare

1. Se inserează un control de tip **`TreeView`**, preluând din **ToolBox / Windows Forms / TreeView** .
2. Se redenumeşte controlul în **`tvCtrl`**.
3. Proprietatea **`Layout / Dock`** a controlului **`TreeView`** ia valoarea `Left`.
4. Pentru ca etichetele nodurilor controlului să poată fi editate la execuţie trebuie ca proprietatea **`Behavior / LabelEdit`** a controlului `TreeView` să ia valoarea `True`.
5. Proprietatea controlului TreeView **`Behavior/Sorted`** ia valoarea `True` pentru a afişa nodurile sortate alfabetic.
6. Se inserează un control de tip ImageList: din **ToolBox / Windows Forms / ImageList**.
7. Cu ImageList selectat, **Properties / Appearance / Images** se adaugă patru imagini corespunzătoare tipurilor de informaţii: produs, reper, material şi una pentru elementul selectat; în acest scop se foloseşte editorul de colecţii, ca în fereastra următoare:



8. Se construieşte meniul aplicaţiei având opţiunile:

- `Operatii`
  - `Salvare date`
  - `Restaurare date`
  - `Ieşire`

9. Se construieşte un meniul contextual având opţiunile:

- `Inserare`
- `Stergere`

10. Se asociază obiectul de tip **`ContextMenu`** la controlul de tip `TreeView`: proprietatea **`Behaviour / ContextMenu / contextMenu1`**.

11. Se tratează opţiunea `Inserare` a context-meniului prin tratarea evenimentului **`Misc/Click`** al opţiunii **`Inserare`** (vezi cod sursă).
12. Se tratează opţiunea **`Ştergere`** a context-meniului prin tratarea evenimentul **`Misc/Click`** pe opţiunea **`Ştergere`**.
13. Se tratează evenimentul **`AfterLabelEdit`** pentru validarea datelor introduse la modificarea etichetei.

Codul sursă, corespunzător celor discutate mai sus, este următorul:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
// namespace-uri pentru IO cu formatari si pentru serializare
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

namespace tree
{
public class Form1 : System.Windows.Forms.Form
{
private System.Windows.Forms.TreeView tvCtrl;
private System.ComponentModel.IContainer components;
private System.Windows.Forms.MainMenu mainMenu1;
private System.Windows.Forms.MenuItem menAlege;
private System.Windows.Forms.MenuItem menSalvare;
private System.Windows.Forms.MenuItem menRestaurare;
private System.Windows.Forms.MenuItem menuItem4;
private System.Windows.Forms.MenuItem menIesire;
private System.Windows.Forms.MenuItem menInserare;
private System.Windows.Forms.MenuItem menStergere;
private System.Windows.Forms.ImageList imageList1;
private System.Windows.Forms.ContextMenu contextMenu1;
private int nrNivele;

private static ArrayList lista;

public Form1()
{
      InitializeComponent();
      nrNivele=0;
}

protected override void Dispose( bool disposing )
{
      if( disposing )
```



```
      {
            if (components != null)
            {
                  components.Dispose();
            }
      }
      base.Dispose( disposing );
}

#region Windows Form Designer generated code

private void InitializeComponent()
{
 this.components = new System.ComponentModel.Container();
 System.Resources.ResourceManager resources =
      new System.Resources.ResourceManager(typeof(Form1));
 this.tvCtrl = new System.Windows.Forms.TreeView();
 this.contextMenu1 = new System.Windows.Forms.ContextMenu();
 this.menInserare = new System.Windows.Forms.MenuItem();
 this.menStergere = new System.Windows.Forms.MenuItem();
 this.imageList1 =
      new System.Windows.Forms.ImageList(this.components);
 this.mainMenu1 = new System.Windows.Forms.MainMenu();
 this.menAlege = new System.Windows.Forms.MenuItem();
 this.menSalvare = new System.Windows.Forms.MenuItem();
 this.menRestaurare = new System.Windows.Forms.MenuItem();
 this.menuItem4 = new System.Windows.Forms.MenuItem();
 this.menIesire = new System.Windows.Forms.MenuItem();
 this.SuspendLayout();
      //
      // tvCtrl
      //
 this.tvCtrl.ContextMenu = this.contextMenu1;
 this.tvCtrl.Dock = System.Windows.Forms.DockStyle.Left;
 this.tvCtrl.HideSelection = false;
 this.tvCtrl.ImageList = this.imageList1;
 this.tvCtrl.LabelEdit = true;
 this.tvCtrl.Location = new System.Drawing.Point(0, 0);
 this.tvCtrl.Name = "tvCtrl";
 this.tvCtrl.RightToLeft =
      System.Windows.Forms.RightToLeft.No;
 this.tvCtrl.SelectedImageIndex = 1;
 this.tvCtrl.Size = new System.Drawing.Size(376, 273);
 this.tvCtrl.Sorted = true;
 this.tvCtrl.TabIndex = 0;
 this.tvCtrl.AfterLabelEdit +=
   new NodeLabelEditEventHandler(this.tvCtrl_AfterLabelEdit);
      //
      // contextMenu1
      //
```

```
this.contextMenu1.MenuItems.AddRange(
     new System.Windows.Forms.MenuItem[]
          { this.menInserare, this.menStergere}     );
     //
     // menInserare
     //
this.menInserare.Index = 0;
this.menInserare.Text = "Inserare";
this.menInserare.Click +=
     new System.EventHandler(this.Inserare);
     //
     // menStergere
     //
this.menStergere.Index = 1;
this.menStergere.Text = "Stergere";
this.menStergere.Click +=
     new System.EventHandler(this.Stergere);
     //
     // imageList1
     //
this.imageList1.ImageSize = new System.Drawing.Size(16, 16);
this.imageList1.ImageStream =
     ((System.Windows.Forms.ImageListStreamer)
     (resources.GetObject("imageList1.ImageStream")));
this.imageList1.TransparentColor =
     System.Drawing.Color.Transparent;
     //
     // mainMenu1
     //
this.mainMenu1.MenuItems.AddRange(
     new System.Windows.Forms.MenuItem[]
          {        this.menAlege}  );
     //
     // menAlege
     //
this.menAlege.Index = 0;
this.menAlege.MenuItems.AddRange(
     new System.Windows.Forms.MenuItem[]
      {
      this.menSalvare,
      this.menRestaurare,
      this.menuItem4,
      this.menIesire
      }            );
this.menAlege.Text = "Operatii";
     //
     // menSalvare
     //
this.menSalvare.Index = 0;
this.menSalvare.Text = "Salvare date";
```



```
this.menSalvare.Click +=
           new System.EventHandler(this.Salvare);
     //
     // menRestaurare
     //
this.menRestaurare.Index = 1;
this.menRestaurare.Text = "Restaurare date";
this.menRestaurare.Click +=
     new System.EventHandler(this.Restaurare);
     //
     // menuItem4
     //
this.menuItem4.Index = 2;
this.menuItem4.Text = "-";
     //
     // menIesire
     //
this.menIesire.Index = 3;
this.menIesire.Text = "Iesire";
this.menIesire.Click +=
     new System.EventHandler(this.Iesire);
     //
     // Form1
     //
this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
this.ClientSize = new System.Drawing.Size(304, 273);
this.Controls.Add(this.tvCtrl);
this.Menu = this.mainMenu1;
this.Name = "Form1";
this.Text = "Form1";
this.ResumeLayout(false);
}
#endregion

[STAThread]
static void Main()
{
      Application.Run(new Form1());
}

private void Inserare(object sender, System.EventArgs e)
{
 TreeNode nodNou = new TreeNode(":");
      // obiect nou de tip TreeNode cu eticheta ":"
 nodNou.ImageIndex = 0;
 if(tvCtrl.Nodes.Count==0 | tvCtrl.SelectedNode==null)
 // nu exista nod in arbore sau nu este selectat nici un nod
 tvCtrl.Nodes.Add(nodNou); //noul nod se adauga pe radacina
 if(tvCtrl.SelectedNode!=null)
 //daca exista un nod selectat in TreeView
```

```
        {
         TreeNode nodSel=tvCtrl.SelectedNode;
         //se preia in nodSel nodul selectat din TreeView
         nrNivele++;
         if(nrNivele>3)
               nrNivele=0;
         nodNou.ImageIndex=nrNivele;
         nodSel.Nodes.Add(nodNou); // adaugare la colectie
        }
   tvCtrl.SelectedNode = nodNou; //nodul adaugat devine selectat
   tvCtrl.ExpandAll(); //se expandeaza nodurile  TreeView-ului
   tvCtrl.Sorted=true;
   nodNou.BeginEdit(); //se initiaza editarea etichetei
}

private void Stergere(object sender, System.EventArgs e)
{
 if(tvCtrl.Nodes.Count>0) //daca exista noduri in arbore
  {
         //se pastreaza nodul selectat in variabila tn
         TreeNode tn=tvCtrl.SelectedNode;
         //daca nodul curent selectat nu e ultimul
         if (tvCtrl.SelectedNode.NextNode!=null)
          //se retine nodul ce succeede nodului curent
               tn=tvCtrl.SelectedNode.NextNode;
         else
               // daca nodul selectat nu este primul nod
               if (tvCtrl.SelectedNode.PrevNode!=null)
                //se retine nodul ce precede nodul curent
                     tn=tvCtrl.SelectedNode.PrevNode;
              //se sterge nodul selectat
         tvCtrl.SelectedNode.Remove();
              // tn devine noul crt selectat
         tvCtrl.SelectedNode = tn;
   }
}

private void Salvare(object sender, System.EventArgs e)
{
 lista=new ArrayList();//lista pentru serializare
 foreach (TreeNode tn in tvCtrl.Nodes)
           //transfer noduri din colectie in lista
        {
           SalvareNod(tn,lista);    //add nod in lista
        }
        // serializare lista in fisierul "date.tree"
 FileStream s =
           new FileStream("date.tree",FileMode.Create);
 BinaryFormatter f =
           new BinaryFormatter();
```



```
 try  { f.Serialize(s, lista);}
 catch (System.Runtime.Serialization.SerializationException)
          {MessageBox.Show("Serializare esuata");}
 s.Close();
}

private void SalvareNod(TreeNode tn, ArrayList lista)
{
 string[] val=tn.Text.Split(':');
 // extragere subsiruri, folosind ':' ca separator
 // pentru a obtine denumirea, respectiv cantitatea
 lista.Add( new
           nod(val[0],Convert.ToInt32(val[1]),
           tn.FullPath,tn.ImageIndex)          );
 // adaugare informatii in lista, preluand din nod
 foreach (TreeNode n in tn.Nodes)
  { //pentru fiecare TreeNode din colectia nodului curent
    SalvareNod(n, lista);
    //apel recursiv, pentru salvarea tuturor nodurilor
  }
}
private void Restaurare(object sender, System.EventArgs e)
{
       rest(tvCtrl,"date.tree");
       //restaurare din fisierul "date.tree"
}

public static void rest(TreeView tvCtrl, string fis)
{
 if (File.Exists(fis))
  {
  // deserializare pentru transfer in ArrayList
       FileStream s = new FileStream(fis,FileMode.Open);
       BinaryFormatter f = new BinaryFormatter();
       try
        {
              lista =(ArrayList) f.Deserialize(s);
        }
       catch
        (System.Runtime.Serialization.SerializationException)
             {
                    MessageBox.Show("Deserializare esuata !!");
                    return;
             }
        s.Close();

       for (int i=0;i<lista.Count; i++)
             //pentru fiecare element al listei
        {
          nod inf =(nod) lista[i];
```

```
        TreeNode nodCurent=
            new TreeNode(inf.val_den+":"+inf.val_cant);
        // se creaza un TreeNode cu Text avand valoarea
        //  denumire:cantitate
        string cn=Convert.ToString(inf.val_cale);
        string[] parti =
            cn.Split(tvCtrl.PathSeparator.ToCharArray());
        // subsirurile existente in numele caii nodului curent
        // se salveaza in variabila parti
        // se foloseste ca separator al sirurilor separatorul
        // controlului TreeView - proprietatea PathSeparator
        if(parti.Length>1)
        // daca exista doua parti, nodul nu este radacina
        {
            TreeNodeCollection nodes =tvCtrl.Nodes;
            // se extrage colectia de noduri a controlului
            TreeNode nodParinte = null;
            CautaNod(parti, ref nodParinte,  nodes);
            // se cauta parintele nodului curent
            if (nodParinte != null)
            //daca nodul curent are parinte
                {
                 nodParinte.Nodes.Add(nodCurent);
                 // adaugare la parintele gasit
                }
        }
        else tvCtrl.Nodes.Add(nodCurent);
        // nodul curent este nod radacina
        // si trebuie adaugat in colectia de noduri a tvCtrl
    }
  }
}


private static void CautaNod( string[] parti,
      ref TreeNode nodParinte,  TreeNodeCollection nodes  )
{
 foreach (TreeNode n in nodes)
 // pentru fiecare nod din colectia de noduri
  {
  if (n.Text.Equals(parti[parti.Length-2].ToString()))
  // daca proprietatea Text a nodului curent coincide
  // cu numele parintelui nodului curent
  nodParinte = n; //se modifica parintele
  CautaNod(parti, ref nodParinte, n.Nodes);
  // apel recursiv pt cautare in colectia de noduri a
  //              nodului curent
  }
}
```



```
private void Iesire(object sender, System.EventArgs e)
{
 Application.Exit();
}


private void tvCtrl_AfterLabelEdit(object sender,
          System.Windows.Forms.NodeLabelEditEventArgs e)
{
 string [] etNoua;
 // validare format general (denumire:cantitate)
 try{etNoua=e.Label.Split(':');}
 //proprietatea Label contine noul text asociat TreeNod-ului
 //daca nu exista ':' in noul text al TreeNod-ului
 catch(System.Exception)
  {
   e.CancelEdit=true; //se suspenda modificarea textului
   return;
  }
  // daca nu s-a respectat formatul general
  // se anuleaza modificarea facuta
  if (etNoua.Length==0) {e.CancelEdit=true; return;}
  if (etNoua.Length==2)
   //textul noii etichete contine un singur caracter ':'
   {
       try{Convert.ToInt32(etNoua[1]);}
        //se incearca convertirea la intreg a  cantitatii
        catch(System.FormatException)
         {  //daca in urma conversiei apare o eroare
            e.CancelEdit=true;
            //se anuleaza modificarea textului
         }
       }
    }
  }
}
```

Pentru serializarea datelor s-a folosit o clasă **nod** cu următorul cod sursă:

```
using System;
using System.Collections;

namespace tree
{
[Serializable()]
```

```
public class nod
  {
      string den;   // denumire prod, mat sau reper
      int cant;     // cantitate prod, mat sau reper
      string cale;  // calea nodului in cadrul structurii
      int idximg;   // indexul listei de imaginii
     public nod(){}

     public nod(string d, int c, string vcale, int vidximg )
           {den=d; cant=c; cale=vcale; idximg=vidximg;}

     public int val_cant
     {
           get   {    return cant;}
           set   {    cant=value; }
     }

     public string val_den
     {
           get   {return den;}
           set   {den=value;}
     }

     public string val_cale
     {
           get   {return cale;}
           set   {cale=value;}
     }

     public int val_idx
     {
           get   {return idximg;}
           set   {idximg=value;}
     }
  }
}
```


LUCRU CU FERESTRE MULTIPLE

1. **Ferestre secundare de dialog**
2. **Aplicații de tip Multiple Document Interface - MDI**

1. Ferestre secundare de dialog

Dialog Box (caseta de dialog) este tot un obiect de tip **Form** care, **prin convenție**, are următoarele particularități:

- proprietatea **FormBorderStyle** este **FixedDialog** (nu poate fi redimensionată la momentul rulării);
- proprietățile **MaximizeBox** și **MinimizeBox** sunt puse pe **False** pentru a preveni minimizarea sau maximizarea ferestrei de dialog; aceste setări elimină automat și butoanele **Maximize** și **Minimize** din meniul sistem (colțul stânga-sus). Eventual se scot complet atât meniul sistem (colțul stânga sus al ferestrei), cât și controalele de minimizare/maximizare setând pe **False** proprietatea **ControlBox** a formei;
- deține cel puțin două butoane, uzual inscripționat **OK** și **Cancel**, pentru închiderea dialogului, cu păstrarea sau nu a modificărilor efectuate prin intermediul dialogului și recepționarea unui mesaj de răspuns.
- Caseta de dialog are și două proprietăți **AcceptButton** și **CancelButton**, prin care se indică butoanele cu rol de **Ok**, respectiv **Cancel**, când butoanele sunt inscripționate altcumva. Totodată fixarea acestei proprietăți asigură posibilitatea terminării dialogului prin tastele **Enter** sau **Esc**, valoarea returnată corespunzând celor două butoane cu care acestea sunt asociate.

Casetele de dialog sunt de două tipuri: **modale** și **nemodale**.

Modal Dialog Box – caseta de dialog modală, nu permite părăsirea ferestrei pentru a accesa altă fereastră a aplicației, înainte de închiderea ei. Se consideră că răspunsul este atât de important încât aplicația nu poate continua până nu se dă acest răspuns.

După crearea formei folosind modelul convențional, forma se activează cu **ShowDialog()**

Modeless Dialog Box – caseta de dialog nemodală, permite părăsirea ei fără să o închizi; astfel se poate reveni ulterior în fereastră pentru a continua lucrul. Chiar când este inactivă, fereastra nemodală se afişează în faţa aplicaţiei părinte, pentru a fi totuşi vizibilă până la închidere (de exemplu, caseta **Find / Replace** într-un editor de text).

Spre deosebire de fereastra de dialog modală, după creare, fereastra de dialog nemodală se activează cu `Show( )`.

Mecanismul de lucru cu ferestre de dialog este simplu. .

1. Se adaugă o nouă formă (**Project / Add Windows Form / Windows Form /** *nume*) şi i se stabilesc proprietăţile unei ferestre de dialog. I se pun două butoane **OK** şi **Cancel** cărora li se fixează proprietatea **DialogResult** pe una din valorile enumerării **DialogResult** (**OK, Cancel, Abort, Retry, Ignore, Yes,** şi **No**) corespunzătoare fiecăruia, după rolul său.

2. Pe forma părinte, legat de Click buton sau de opţiune meniu se instanţiază şi activează fereastra de dialog:

```
private void button1_Click
                    (object sender, System.EventArgs e)
  {
    macheta m1 = new macheta();
    m1.ShowDialog();
    if(m1.DialogResult == DialogResult.OK)
        Camp.Text = m1.Camp.Text;
    else
        f1Camp.Text = "";
  }
```

Tot în acestă funcţie se tratează modul de terminare a dialogului, decizând dacă se preiau (`DialogResult.OK`) sau nu (celelalte posibile variante returnate `Cancel, Abort, Retry, Ignore, Yes,` şi `No`) valorile introduse în diverse câmpuri, pe durata afişării dialogului.

Se observă că această funcţie asigură vizibilitate simultană atât asupra câmpurilor din forma 1, cât şi a câmpurilor din forma 2. Cele din forma 1 sunt văzute implicit deoarece funcţia de tratare aparţine acestei clase, iar cele din forma 2 se califică pe baza obiectului de tip forma 2 instanţiat aici şi care este vizibil în tot blocul de definire.

Aşadar, după închidere şi până la ieşirea din blocul funcţiei care a activat dialogul, obiectul dialog încă mai există şi pot fi preluate proprietăţi ale acestuia.

Butoanele puse pe forma 2 vor inchide automat forma (nu trebuie dat explicit `Close()`), dacă au setată proprietatea **DialogResult** pe orice valoare din enumerare, exceptând `None`.

La închidere, proprietatea **DialogResult** a formei preia automat valoarea **DialogResult** a butonului care a închis dialogul, astfel încât să ştim după dispariţia imaginii dialogului ce buton a cauzat terminarea lui şi să reacţionăm în consecinţă. Mai mult decât atât, proprietatea **DialogResult** a formei nici măcar nu trebuie preluată, căci ea este returnată şi de funcţia de `ShowDialog()` la închiderea dialogului.

Acum ne putem explica simplitatea lucrului cu controale de dialog de tip **FontDialog, ColorDialog, OpenFileDialog, SaveFileDialog** etc.

Acestea deţin proprietăţi specifice prin intermediul cărora se pot prelua la închiderea dialogului, informaţiile introduse sau alese pe durata derulării dialogului.

ColorDialog prin selectarea culorii îşi setează propria sa proprietate **Color**, pe care la întoarcere în apelant o putem prelua drept culoare pentru vreuna din proprietăţile unui control de pe forma principală:

```
label1.BackColor = colorDialog1.Color;
```

Similar, **FontDialog** întoarce fontul selectat în propria sa proprietate **Font**:

```
label1.Font = fontDialog1.Font;
```

OpenFileDialog are proprietatea **FileName** ce va fi încărcată cu numele fişierului selectat la vizualizarea controlului de tip dialog:

```
OpenFileDialog openFileDialog2= new OpenFileDialog();
openFileDialog2.ShowDialog();
MessageBox.Show("Ati ales fisierul " +
                        openFileDialog2.FileName);
```

Exerciţiu

Exemplificaţi utilizarea controalelor **OpenFileDialog** şi **SaveFileDialog**, construind o aplicaţie gen **Notepad**, specializată în editarea, salvarea şi restaurarea unui text în / din fişier.

1. Pe o formă dimensionată corespunzător, se adaugă un **TextBox** numit **editData**; i se pune proprietatea **Multiline** pe **True** şi va fi docat **Fill**, astfel încât să umple întreaga formă.
2. Se adaugă în antet `using System.IO;` pentru recunoaşterea obiectelor de I/O.
3. Se aduce din ToolBox un meniu căruia i se adaugă pe **Fisier** opţiunile **Open**, **Save** şi **Close**.
4. Se declară un membru în clasa Form1 **public Stream fisier;**
5. Se tratează opţiunea **Open** cu funcţia:

```
private void menOpen_Click(object sender, System.EventArgs e)
{
        OpenFileDialog openFileDialog2= new OpenFileDialog();
        if(openFileDialog2.ShowDialog()==DialogResult.OK )
        {
                //MessageBox.Show("Ati ales fisierul " +
                //  openFileDialog2.FileName);
                fisier = openFileDialog2.OpenFile();
                StreamReader cititor = new StreamReader(fisier);
                //StreamReader cititor =
                //           new StreamReader("c:\\myfile.txt");

                string lin;int i=0; editData.Text="";
                while((lin =cititor.ReadLine())!=null)
                {
                  //nu: editData.Lines.SetValue(lin,i);
                  //nu: editData.Lines[i].Insert(0,lin);
                  editData.Text+=lin+"\r\n";
                      // !! ca sa poata adauga si in regim de
                      //  editare cu Enter
                  //nu: editData.Lines[i]=lin;
                }
                cititor.Close();
        }
}
```

Respectând convenţia, **OpenFileDialog** returnează ca orice dialog, proprietatea **DialogResult**.

Dacă pe durata afişării dialogului `OpenFileDialog` s-a selectat un fişier, iar la închidere s-a dat **Ok**, se poate deschide fişierul ales, folosind funcţia **OpenFile()** a aceluiaşi dialog. De remarcat că dialogul în sine nu face decât selectarea unui fişier, nu îl şi deschide cum am putea crede după numele controlului. Controlul ţine în schimb numele fişierului, astfel încât metoda sa `openFileDialog2.`**OpenFile()** să nu mai necesite parametri de intrare.

Fişierului i se ataşează apoi un `StreamReader`, cu care se citeşte linie cu linie conţinutul fişierului şi se adaugă TextBox-ului.

Se observă că adăugarea se face cu operatorul += la proprietatea `Text` a controlului; la întâlnirea unui `"\r\n"` textul va fi stocat în alt string al vectorului **editData.Lines**.

StreamReader şi **StreamWriter** sunt recomandaţi pentru fişiere despre a căror structură se ştie ceva (terminatorul de înregistrare este ***newline***).

6. Se tratează opţiunea **Save** cu funcţia:

```
private void menSave_Click(object sender, System.EventArgs e)
{
        SaveFileDialog saveFileDialog2= new SaveFileDialog();
        if( saveFileDialog2.ShowDialog()==DialogResult.OK)
        {
                //MessageBox.Show("Ati ales fisierul "+
                //              saveFileDialog2.FileName);
                fisier = saveFileDialog2.OpenFile();
                StreamWriter scriitor = new StreamWriter(fisier);
                foreach( string lin in editData.Lines)
                                scriitor.WriteLine(lin);

                scriitor.Close();
        }
}
```

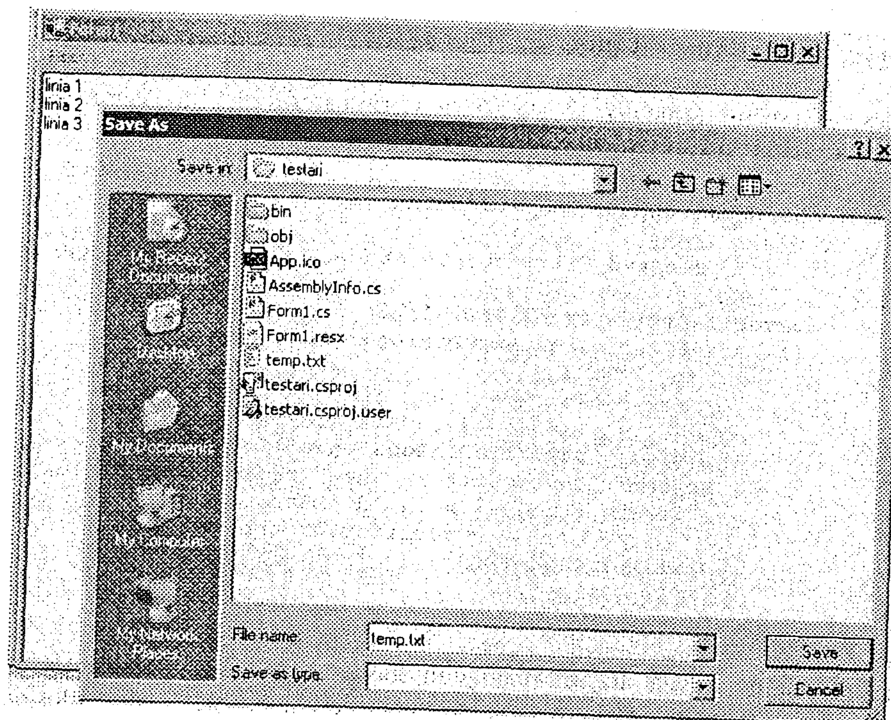
Dacă pe durata afişării dialogului **SaveFileDialog** s-a selectat un fişier, iar la închidere s-a dat **Ok**, se poate deschide fişierul ales folosind funcţia **OpenFile()** a aceluiaşi dialog; fişierului i se ataşează apoi un `StreamWriter`, care ulterior preia toate liniile din vectorul de string-uri **editData.Lines** al textBox-ului şi le scrie în fişier.

7. Se asociază evenimentului `Click` pe opţiunea **Close** funcţia:

```
private void menClose_Click
                (object sender, System.EventArgs e)
        {
                try { fisier.Close();}
                catch { }
                editData.Text="";
        }
```

Prin **try** funcţia maschează eventuala tentativă de închidere a unui fişier deja închis. De observat că dacă la `Save` nu se dă `OK` fişierul rămâne deschis.

Clasa **FileInfo** ataşată unui fişier returnat de OpenFileDialog permite extragerea selectivă a unor informaţii despre fişier: numele, extensia etc.



Lăsăm ca exerciţiu completarea aplicaţiei cu butoane pe un ToolBar, ca alternativă pentru apelul funcţiilor de mai sus, sau schimbarea proprietăţilor textului (font, dimensiune, culoare etc.).

Tot ca exerciţiu, să se transforme apoi controlul într-un control de utilizator **EditBox** şi să fie înscris în ToolBox.

2. Aplicaţii de tip Multiple Document Interface - MDI

Din punct de vedere al mulţimii datelor cu care lucrează o aplicaţie, s-au desprins două arhitecturi, bine conturate în majoritatea bibliotecilor de clase utilizate în programarea orientată obiect:

- arhitectura **SDI** – Single Document Interface
- arhitectura **MDI** – Multiple Document Interface

Ele se bazează pe ideea că datele sunt de obicei stocate în clase « document » specializate şi că o aplicaţie poate lucra cu unul sau mai multe documente. Scopul arhitecturii MDI este de a permite utilizatorului să lucreze simultan cu mai multe documente fără să lanseze o nouă instanţă a aplicaţiei, ci doar să deschidă o nouă fereastră în cadrul aceleeaşi aplicaţii.

Vom deosebi în acest caz o formă (sau fereastră) cadru **principală** sau **părinte**, cu meniu propriu şi o formă (sau fereastră) cadru **secundară** sau **copil**, ce poate fi **instanţiată în mai multe exemplare**, câte unul pentru fiecare document deschis la un moment dat. Forma secundară poate avea sau nu propriul meniu, sau poate mixa opţiuni cu cele din meniul formei principale.

Legătura dintre cele două tipuri de ferestre se manifestă şi vizual, în sensul că **formele care au părinte pot fi afişate numai în interiorul ferestrei părinte.**

Mecanismul este destul de simplu: obiectele Form au două proprietăţi

- **IsMdiContainer,** care pusă pe **true** anunţă că forma este o fereastră principală ce poate ţine ferestrele altor forme, secundare.
- **MdiParent** care pentru o formă secundară ţine referinţa formei părinte.

În acest fel formele pot fi legate unele de altele.

```
private void menFerNoua_Click(object sender, EventArgs e)
{
    Form2 f2= new Form2();
    f2.MdiParent = this;
    f2.Show();
}
```

De reţinut că Form1 are setată vizual proprietatea **IsMdiContainer**, în timp ce forma secundară Form2 are setată proprietatea **MdiParent** prin cod, nefiind browsabilă.

În general, aplicaţiile MDI presupun stocarea datelor din documente în fişiere şi respectiv încărcarea datelor din fişiere în documente, prin procedeul de serializare. Din acest motiv, exemplificarea o vom face în paralel cu lucru cu fişiere.

Exercițiu

Să se construiască o aplicație MDI cu vizualizare tip formular, care va conține la rândul ei mai multe ferestre copil ce permit **preluarea dintr-o machetă de introducere date** a unor informații despre materiale.

Aplicația asigură și **serializarea / deserializarea** datelor la nivel de articol.

Rezolvare

1. Se construiește un proiect *Visual C#* cu aplicație *Windows Application*;
2. Fereastra cadru principală a aplicației trebuie să fie container pentru ferestrele copil: proprietatea `IsMdiContainer` a formei se setează pe `true`;
3. Se adaugă un meniu ferestrei cadru principale a aplicației, pentru manipularea ferestrelor copil, având opțiunile:
   - `Fişier`
     - `Fereastră nouă`
     - `Închide toate ferestrele`
     - `Ieşire`
     - **`Aranjare ferestre`**
       - `În cascadă`
       - `Orizontal`
       - `Vertical.`

   În acest scop se aduce din **Toolbox** un control **MainMenu** și i se stabilesc opțiunile de mai sus. Se asociază apoi meniul cu fereastra principală: proprietatea **Menu** a formei principale (`Form1`) ia valoarea `mainMenu1`.
4. Se crează șablonul ferestrei cadru copil –fereastra de interacțiune cu documentul aplicației, inserând o formă nouă: **Project / Add Windows Form / Windows Form** și i se dă un nume.
5. Se adaugă ferestrei copil un meniu cu opțiuni care țin de manipularea datelor: `Date` cu suboptiunile `Incarcă` și `Salvează`.
6. Pe suboptiunea `Fereastră nouă` a meniului ferestrei principale, se instanțiază o nouă fereastră copil:

```
//se creaza o noua instanta a formei copil
```



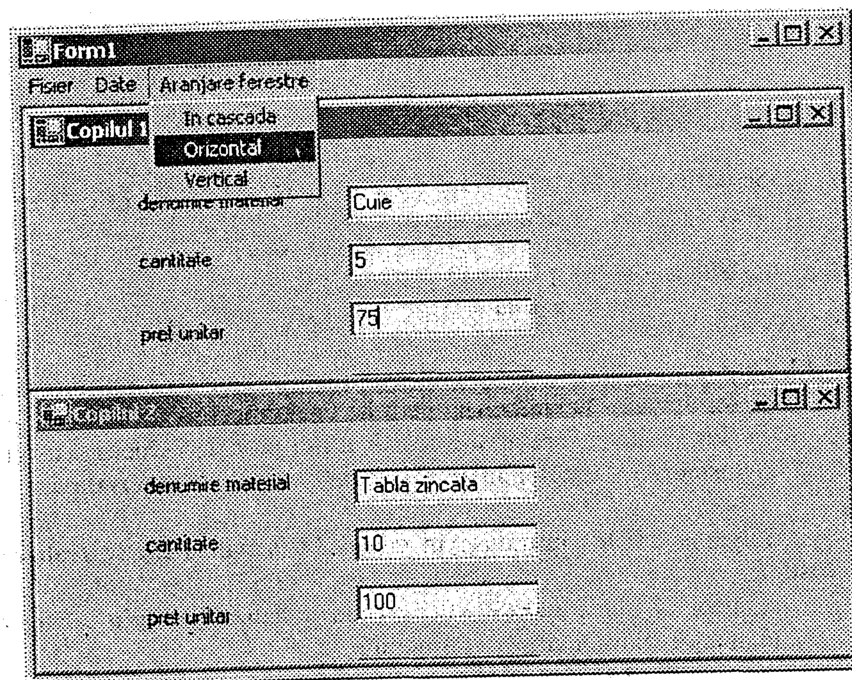
```
        Form2 fcopil = new Form2();
        fcopil.MdiParent=this; //se stabileste parintele
//forma copil se aduce la dimensiunile formei parinte
fcopil.ClientSize = this.ClientSize;
nrc++; //nrc - numărul formelor copil create;
        // inițializat cu 0 în constructorul formei 1

fcopil.Text="Copilul " + nrc; // titlul formei copil
fcopil.Show(); //se afiseaza forma copil
```

7. Pentru aranjarea ferestrelor copil, în funcție de opțiunea selectată a meniului `Aranjare`, se folosește metoda **LayoutMdi** a formei părinte și membrii enumerației `MdiLayout`: `ArrangeIcons`, `Cascade`, `TileHorizontal`, `TileVertical`.

   Pe opțiunea `Orizontal` a meniului `Aranjare ferestre`, vom scrie:

```
this.LayoutMdi(MdiLayout. TileHorizontal);
```



Se procedează similar pentru celelalte opțiuni de aranjare a ferestrelor.

8. Cele trei opțiuni principale ale meniului compus sunt: `Fişier`, `Aranjare ferestre`, `Date`. Dar în aplicațiile Windows este

obişnuită poziţionarea meniului **Aranjare ferestre** la urmă. Pentru a schimba ordinea: proprietatea **MergeOrder** a opţiunii `Date` a meniului i se schimbă valoarea în **1**, iar a opţiunii **Aranjare ferestre** în **2**, opţiunea **Fişier** rămânând cu valoarea **0**.

9. Se adaugă, în forma cadru copil (*Forma2*), controale (`Label` şi `TextBox`) pentru introducerea de date pentru materiale: denumire, cantitate, pret unitar şi **pentru afişarea valorii** materialului (controlul de tip `TextBox` corespunzător are proprietatea **Enabled** pe valoare **false**).

10. Se inserează în proiect o nouă clasă **material**: **Project / Add Class / Class / material**, având variabilele private: `den, cant, pret, val` şi proprietăţile accesabile prin *get* şi *set:* `val_den, val_cant, val_pret, val_val`.

11. Odată cu modificarea valorilor `cant` sau `pret`, în fereastra copil (evenimentul **TextChanged**), se modifică corespunzător şi valoarea `val`:

```
double t=Convert.ToDouble(tb_pret.Text)*
                    Convert.ToInt32(tb_cant.Text);
tb_val.Text=Convert.ToString(t);
```

12. Se defineşte şi se instanţiază un obiect **mat_meu** de tip **material**:

- în clasa `Form2` se declară variabila:

```
        private material mat_meu;
```

- iar în constructorul `Form2()`:

```
        mat_meu = new material();
```

13. Se pot salva datele obiectului **mat_meu**, într-un fişier, prin operaţia de serializare. **Serializarea** permite unui obiect să fie convertit într-un flux de date, care apoi este salvat într-un fişier. Operaţia inversă, de restaurare a datelor anterior salvate, se numeşte **deserializare**.

Salvarea datelor introduse în controale (serializarea) se realizează astfel:

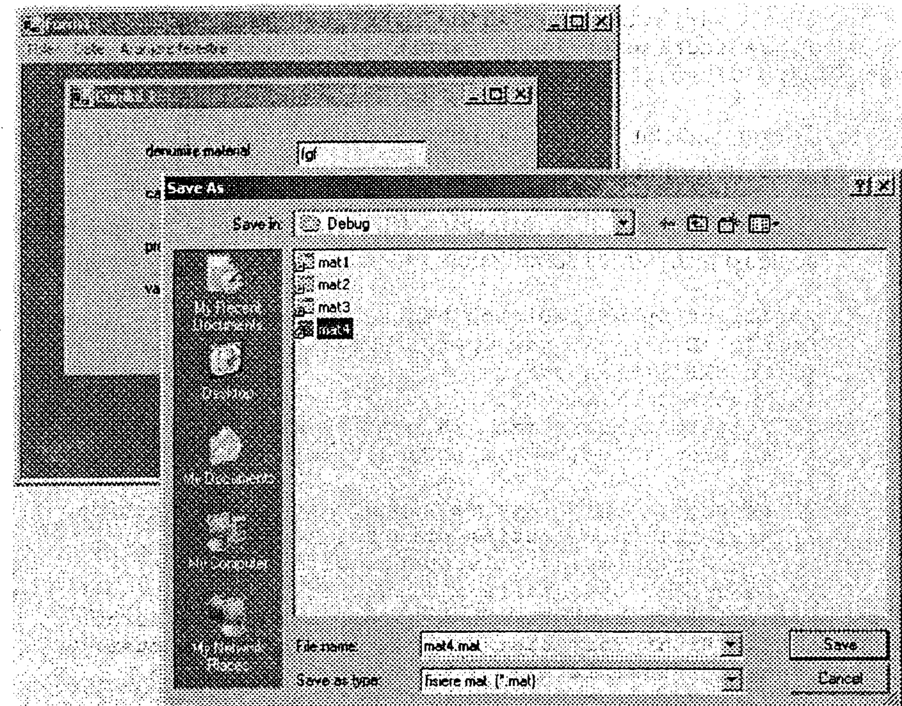
- se declară clasa **material** ca fiind serializabilă, prin adăugarea înaintea definiţiei clasei a atributului:

```
[Serializable()]
```

- se adaugă cu **using** namespace-urile:

```
System.IO;
System.Runtime.Serialization.Formatters.Binary;
```


- se realizează salvarea datelor introduse în fereastra copil activă, într-un fişier de tip **.mat,** astfel:




```
// conversiile datelor introduse în TextBox-uri,

        mat_meu.val_den=tb_den.Text;
        mat_meu.val_cant=Convert.ToInt16(tb_cant.Text);
        mat_meu.val_pret=Convert.ToDouble(tb_pret.Text);
        mat_meu.val_val=Convert.ToDouble(tb_val.Text);

 // alegerea numelui fişierului în care se vor salva

        SaveFileDialog fd = new SaveFileDialog();
        fd.CheckPathExists=true;
        fd.Filter="fisiere mat  (*.mat)|*.mat";
        if(fd.ShowDialog()==DialogResult.OK)
        {
        //se crează un Stream, folosit ca depozit de date

        Stream myfs = File.Create(fd.FileName);
        // se instanţiază un BinaryFormatter folosit
```

```
        // pentru formatarea datelor, în  acest caz un
        //                  flux binar
        BinaryFormatter serializer =
                        new BinaryFormatter();
        // obiectul este scris în flux
        serializer.Serialize(myfs, mat_meu);
    // se închide streamul pentru a finaliza serializarea
    myfs.Close();
        }
```

În fişierul selectat / creat la salvare, există o instanţă a obiectului **mat_meu**, în format binar.

14. Operaţia inversă, de restaurare (**deserializare**) se execută după aceleaşi principii ca serializarea:

```
// obiect OpenFileDialog pentru selectare fişier *.mat
OpenFileDialog fd=new OpenFileDialog();
fd.CheckFileExists=true;
fd.CheckPathExists=true;
fd.Filter="fisiere mat  (*.mat)|*.mat";
if(fd.ShowDialog()==DialogResult.OK)
{
// se crează un obiect de tip Stream şi se deschide
// fişierul care conţine obiectul serializat
        Stream myfs= File.OpenRead(fd.FileName);
// se crează o instanţă a clasei folosită la
// serializare pentru formatarea obiectului salvat
        BinaryFormatter deser= new BinaryFormatter();
// metoda Deserialize  a obiectului folosit la
// formatare reconverteşte fluxul de date într-un
// obiect de tip material

        mat_meu=(material)(deser.Deserialize(myfs));
        myfs.Close();
// se realizează conversia în string pentru a putea fi
// încarcate datele în TextBox-uri
        tb_den.Text = mat_meu.val_den;
        tb_cant.Text=Convert.ToString(mat_meu.val_cant);
        tb_pret.Text=Convert.ToString(mat_meu.val_pret);
        tb_val.Text =Convert.ToString(mat_meu.val_val);
}
```

Pentru simplificare am presupus aici o serializare la nivel de înregistrare; în mod curent înregistrările din machetă sunt preluate una câte una într-o colecţie `ArrayList`, care este la rândul ei serializată / deserializată în întregime, dintr-un singur apel `Serialize / Deserialize` (a se vedea aplicaţia de serializare a unei facturi din subcapitolul **Gestiunea unitară a controalelor unei forme**) .

SUPORTUL .NET PENTRU IMPRIMARE ŞI PREVIZUALIZARE

1. **Clasa PrintDocument**
2. **Previzualizarea documentului de imprimat**
3. **Imprimarea documentelor utilizând generatorul de rapoarte *CrystalReports***

Namespace-urile **System.Drawing.Printing** şi **System.IO** furnizează definiţiile pentru clasele si metodele de imprimare şi se dau de obicei sub forma:

```
using System.Drawing;
using System.Drawing.Printing;
using System.IO;
```

1. Clasa PrintDocument

Imprimanta este o locaţie care dispune de un obiect Graphics ce poate fi folosit la scriere. Clasa **PrintDocument** gestionează procesul de imprimare; ea poate fi văzută ca o conexiune dintre aplicaţie şi imprimantă.

PrintDocument furnizează totodată un obiect reutilizabil ce transmite ieşirea la imprimantă; clasa conţine şi metoda **Print** – pentru startarea procesului de imprimare.

Evenimente sesizate:

- `BeginPrint` – lansat înaintea imprimării primei pagini din document ;
- `EndPrint` – declanşat la sfârşitul imprimării ultimei pagini din document ;
- `PrintPage` – activat la începutul fiecărei pagini;
- `QueryPageSettings` - lansat imediat înainte de `PrintPage`, pentru a stabili particularităţile de imprimare ale paginii curente.

Schematic, **mecanismul de imprimare** este următorul:

- aplicaţia declară un membru de clasă `PrintDocument` (sau aduce un control `PrintDocument` din ToolBox);

- legat de un buton sau de o opţiune meniu, aplicaţia crează o instanţă a clasei `PrintDocument` şi-i invocă metoda **`Print();`**
- apelul acestei metode produce evenimentul **`PrintPage;`**
- acţiunea de desenare constă de fapt în ceea ce programatorul pune ca instrucţiuni în handler-ul de tratare a evenimentului **`PrintPage`**. De observat că handler-ul pentru `PrintPage` primeşte automat ca argument un obiect **`PrintPageEventArgs`**, care dispune de proprietatea **`Graphics`** ce este o referinţă la contextul grafic al imprimantei.

Exemplu simplu.

- Se aduc din ToolBox `mainMenu1, textBox1, printDocument1;`
- meniului i se pune opţiunea `Imprima`, tratată pe *Click* printr-o funcţie ce apelează `printDocument1.Print();`
- `textBox1.Text` se stabileşte în **Properties**: "Exemplu de text scris la imprimanta"
- lui `printDocument1` i se tratează evenimentul `PrintPage` cu o funcţie care conţine:

```
e.Graphics.DrawString
(     textBox1.Text,
      new Font("Arial",12),
      new SolidBrush(Color.Red),
      printDocument1.DefaultPageSettings.Margins.Left,
      printDocument1.DefaultPageSettings.Margins.Top
);
```

Trebuie să avem instalată o imprimantă pentru a vedea că primeşte documentul.

Dacă se aduce din ToolBox şi un **`PrintPreviewDialog`** şi i se pune proprietatea `Document` pe `printDocument1`, adică pe controlul existent pe formă, putem declanşa pe Click opţiune `Previzualizare` din meniul `Fişier`, afişarea dialogului:

```
private void menuItem3_Click
      (object sender, System.EventArgs e)
  {
          printPreviewDialog1.ShowDialog();
  }
```


Afişarea dialogului antrenează automat şi metoda `Print` a documentului asociat acestuia, care la rândul ei produce evenimentul `PrintPage`, care este deja tratat, numai că ieşirea e reorientată acum spre ecran, nu spre imprimantă.

Stabilirea proprietăţilor paginii şi imprimantei

1. Dacă se doreşte şi **modificarea setărilor de pagină şi / sau imprimantă** înainte de imprimare, se vor adăuga la formă încă două controale specializate: **`pageSetupDialog1`** şi **`printDialog1`**. Pe meniul `Fişier` se adaugă opţiunile `PageSetup` şi `PrinterSetup`.
2. Se pune `using System.Drawing.Printing;` dacă se doreşte calificarea simplă a obiectelor şi metodelor din namespace-ul cuprinzând funcţiile de imprimare.
3. Se scrie handler-ul de tratare a opţiunii **`PageSetup`** (se presupune că am creat şi instanţiat deja obiectul `pageSetupDialog1` la nivel de formă):

```
private void menPageSetup_Click
                  (object sender, System.EventArgs e)
 {
      pageSetupDialog1.PageSettings = new PageSettings();
      if(pageSetupDialog1.ShowDialog() == DialogResult.OK)
      {
        printDocument1.DefaultPageSettings =
                  pageSetupDialog1.PageSettings;
      }
 }
```

După cum se observă, referinţa către un obiect **`PageSettings`** trebuie mai întâi iniţializată cu un obiect instanţiat adhoc, care mai apoi va primi valorile setate de către utilizator în dialogul de Page Setup ; aceste valori sunt preluate, la închiderea dialogului, în setările default ale documentului.

4. Se scrie handler-ul de tratare a opţiunii **`PrintSetup`** din meniu (există obiectul `printDialog1` la nivel de formă, deja instanţiat) :

```
private void menPrintSetup_Click
                  (object sender, System.EventArgs e)
 {
      printDialog1.PrinterSettings = new PrinterSettings();
      if(printDialog1.ShowDialog() == DialogResult.OK)
```

```
    {
      printDocument1.PrinterSettings =
                    printDialog1.PrinterSettings;
    }
}
```

Maniera de lucru este aceeaşi: se instanţiază un obiect `PrinterSettings` legat prin referinţă de dialogul `printDialog1`; la închiderea dialogului, valorile introduse de utilizator în câmpurile de editare ale dialogului se regăsesc stocate în obiectul instanţiat mai sus şi vor fi preluate de documentul care aşteaptă pentru imprimare.

Exerciţiu

Să se printeze rezultatul unei interogări pe o bază de date. Să se asigure imprimarea informaţiilor pe două coloane.

Exerciţiu

Să se scrie aplicaţia care citeşte conţinutul unui fişier text din disc şi-l imprimă. Se va scrie cod sursă, în locul aducerii controalelor din ToolBox.

Rezolvare

- Se declară în clasa aplicaţiei un obiect `public PrintDocument pd;` adăugarea se poate face şi cu MouseRight pe clasa formular;
- Se pun şi se tratează pe meniul principal opţiunea `Fisier`, cu subopţiunile de `Open` şi `Close`, pentru fişierul text de imprimat / previzualizat.

```
private void Deschide(object sender, System.EventArgs e)
{
    try { fisier = new StreamReader("C:\\MyFile.txt"); }
    catch(Exception ex) {  MessageBox.Show(ex.Message); }
}

private void Inchide(object sender, System.EventArgs e)
    {
        fisier.Close();
    }
```

`fisier` este un obiect de tip `StreamReader` declarat membru al clasei formular `Ex_Print`.

Observaţie.

S-ar fi putut invoca un control **OpenFileDialog,** respectiv **SaveFileDialog,** pentru generalizare

- Pe un buton sau tot pe o opţiune (`Imprima`) din meniul `Fisier` se instanţiază un obiect `printDocument`, se anunţă funcţia de tratare a evenimentului `PrintPage` al documentului şi se lansează metoda pentru imprimarea câte unei pagini:

```
private void printButton_Click(object sender, EventArgs e)
 {
     try
        {
              printFont = new Font("Arial", 10);
              // instantiere font declarat in forma
              pd = new PrintDocument();
              pd.PrintPage += new
                 PrintPageEventHandler(this.pd_PrintPage);
              pd.Print();
        }
     catch(Exception ex)
      { MessageBox.Show("Fisier nedeschis: "+ex.Message); }

 }
```

- Funcţia de mai sus e declarată automat ca funcţie de tratare a evenimentului click de mouse, dacă cu `Designer`-ul am stabilit în **Properties / Events** acest lucru; dacă nu am lucrat cu `Forms Designer`, această legătură se va face manual, sub forma:

```
this.printButton.Click +=
          new System.EventHandler(this.printButton_Click);
```

Ultima comandă din funcţie, `pd.Print();` iniţiază procesul de imprimare, care lansează la rândul lui funcţia `pd_PrintPage`, pentru fiecare pagină de imprimare; acest lucru se întâmplă deoarece cu o linie mai sus am declarat pentru documentul de imprimare ***pd***, funcţia `pd_PrintPage` drept funcţie de tratare a evenimentului `PrintPage`, (prin apelul `PrintPageEventHandler(pd_PrintPage)`):

```
pd.PrintPage += new PrintPageEventHandler(this.pd_PrintPage);
```

• Funcţia de tratare ar putea arăta astfel:

```
private void pd_PrintPage
                    (object sender, PrintPageEventArgs ev)
{
  float liniiPePagina = 0;
  float yPoz = 0;   int nr = 0;
  float stgMargine = ev.MarginBounds.Left;
  float susMargine = ev.MarginBounds.Top;
  string linie_txt = null;

  // Calcul numar de linii / pagina
  liniiPePagina = ev.MarginBounds.Height /
                      printFont.GetHeight(ev.Graphics);

  // Imprima linie cu linie din fisier
  while(nr < liniiPePagina &&
                ((linie_txt = fisier.ReadLine()) != null))
   {
     yPoz =
       susMargine + (nr * printFont.GetHeight(ev.Graphics));
     ev.Graphics.DrawString
             (linie_txt, printFont, Brushes.Black,
                 stgMargine, yPoz, new StringFormat()  );
     nr++;
   }

   // in cazul in care mai exista inca linii in fisier
   if(linie_txt != null)
            ev.HasMorePages = true;
   else     ev.HasMorePages = false;
 }
```

Se observă că `MarginBounds` fiind dreptunghiul de imprimare, el permite calculul numărului de linii pe o pagină pornind de la înălţimea dreptunghiului şi înălţimea fontului. Există mai multe forme supraîncărcate ale funcţiei **`DrawString`**; aici s-a optat pentru a da şirul de scris, fontul, pensula, poziţia de scris sub forma a două valori `float` şi un format implicit de conversie a şirului.

Poziţia fiecărei linii s-a calculat în funcţie de marginea de sus a paginii, numărul liniei de scris şi înălţimea unei linii (depinzând de înălţimea fontului).

Variabila booleana `ev.`**`HasMorePages`** solicită sau nu relansarea automată a procesului de imprimare a unei pagini, în funcţie de existenţa sau inexistenţa altor linii neimprimate încă.

2. Previzualizarea documentului de imprimat

Pentru partea de previzualizare, lucrurile stau aproximativ la fel. Este nevoie de un obiect **`PrintPreviewDialog`** care se poate aduce din Toolbox; se crează astfel automat un obiect **`printPreviewDialog1`**

```
private PrintPreviewDialog printPreviewDialog1;
```

ale cărui metode pot fi invocate pentru activarea dialogului şi previzualizarea paginilor de imprimat.

Diferenţa între a trage din Toolbox sau a declara un obiect prin program în clasa formei constă în faptul că `Forms Designer` pune el declaraţia referinţei şi instanţierea obiectului, iar selectând obiectul plasat în partea de jos a formei, putem cu **Properties**, adăuga direct funcţia de tratare a evenimentului `PrintPage`:

Prin fixarea proprietăţii **`Document`** a `printPreview`-ului pe obiectul `printDocument` **`pd`**, acesta din urmă va considera că suportul de afişare nu mai este imprimanta, ci chiar dialogul de previzualizare.

Pe Click opţiune de meniu se instanţiază obiectul de tip `PrintDocument` şi i se `încarcă` delegatul asociat evenimentului `PrintPage` cu metoda furnizată de noi:

```
private void opt_previewClick
                     (object sender, System.EventArgs e)
     {
           pd = new PrintDocument();
           pd.PrintPage += new PrintPageEventHandler
                            ( this.preview_PrintPage );
           printPreviewDialog1.Document = pd;
           printPreviewDialog1.ShowDialog();
     }
```

La vizualizarea dialogului **se declanşează automat şi metoda** **`Print`** a documentului, care declanşează la rândul ei evenimentul `PrintPage`, tratat de noi prin funcţia `preview_PrintPage`.

```
private void preview_PrintPage
                    (object sender, PrintPageEventArgs ev)
 {
      String linie_txt = "Text de scris in document";

      Font fnt = new Font("Lucida Handwriting", 16);
```

```
SolidBrush pns = new SolidBrush(Color.Black);

ev.Graphics.FillRectangle(Brushes.Red,
                new Rectangle(200, 200, 100, 100));
    // Colt stanga-sus
float x = 150.0F;          float y = 150.0F;

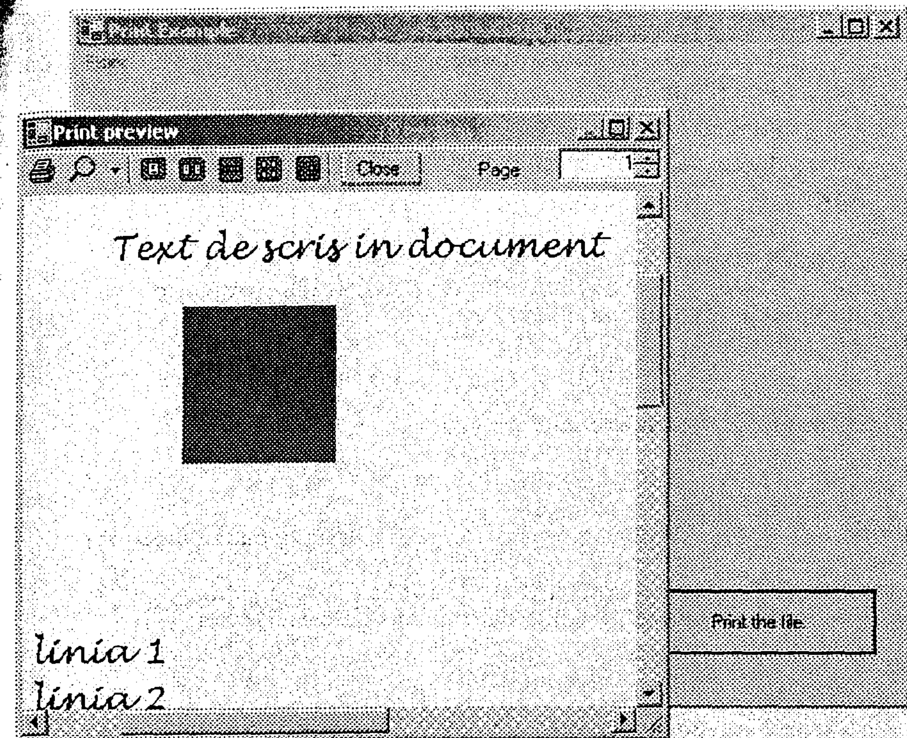
ev.Graphics.DrawString(linie_txt,fnt,pns,x,y);
ev.HasMorePages = true;
float liniiPePagina = 0;
float yPoz = 0;            int nr =10;
float stgMargine = ev.MarginBounds.Left;
float susMargine = ev.MarginBounds.Top;
linie_txt = null;

// Calcul numar de linii / pagina.
liniiPePagina = ev.MarginBounds.Height /
            fnt.GetHeight(ev.Graphics);
// Vizualizeaza linie cu linie
while(nr < liniiPePagina &&
            ((linie_txt=fisier.ReadLine()) != null))
 {
 yPoz = susMargine + (nr * fnt.GetHeight(ev.Graphics));
 ev.Graphics.DrawString(linie_txt, fnt, Brushes.Black,
            stgMargine, yPoz, new StringFormat());
 nr++;
 }

// cazul in care exista mai multe linii in fisier
if(linie_txt != null)
            ev.HasMorePages = true;
else        ev.HasMorePages = false;

}
```

Puteam foarte bine să asociem aceeaşi funcţie de tratare a evenimentului `PrintPage` folosită şi la imprimarea propriu-zisă; aici am urmărit însă funcţionarea de sine stătătoare a previzualizării, independent de imprimare.



Transformări în pagina de imprimare

Contextul grafic al imprimantei permite şi **scrierea sub diverse unghiuri a textului**; în acest scop el dispune de metoda `RotateTransform()`:

Unghiul se dă în grade, în sens invers trigonometric

Există de asemenea funcţii pentru **translaţii** **`TranslateTransform()`** şi **scalări** **`ScaleTransform()`**. Combinarea lor permite obţinerea de efecte de genul:

- trasarea unor figuri complexe, cu etichetări sub diverse unghiuri;
- mirroring sau imprimarea "în oglindă" necesară la imprimarea pe foaie de calc a negativului tipografic.
- scrierea răsturnată, când foaia de imprimare având un colţ îndoit, trebuie introdusă invers.

Efectele trebuie calculate atent, deoarece:

- ele se cumulează la mai multe apeluri succesive pentru oricare din cel trei funcţii amintite;
- schimbând unghiul, se schimbă şi direcţia axelor ox şi oy şi implicit interpretarea valorilor curente pentru abscisa şi ordonata punctului de scriere.

```
RectangleF cadru=new RectangleF
                (500,500,300,fnt.GetHeight(ev.Graphics));
Pen cr=new Pen(Brushes.Black,2);

for(int k=0; k< 3; k++)
 {
  ev.Graphics.RotateTransform(20*k);
  cadru.X+=100*k; cadru.Y-=100*k;

  ev.Graphics.DrawRectangle
          (cr,cadru.X,cadru.Y,cadru.Width,cadru.Height);

  ev.Graphics.DrawString(""+k+" Rotire text", fnt,
          Brushes.Black, cadru, new StringFormat());
 }

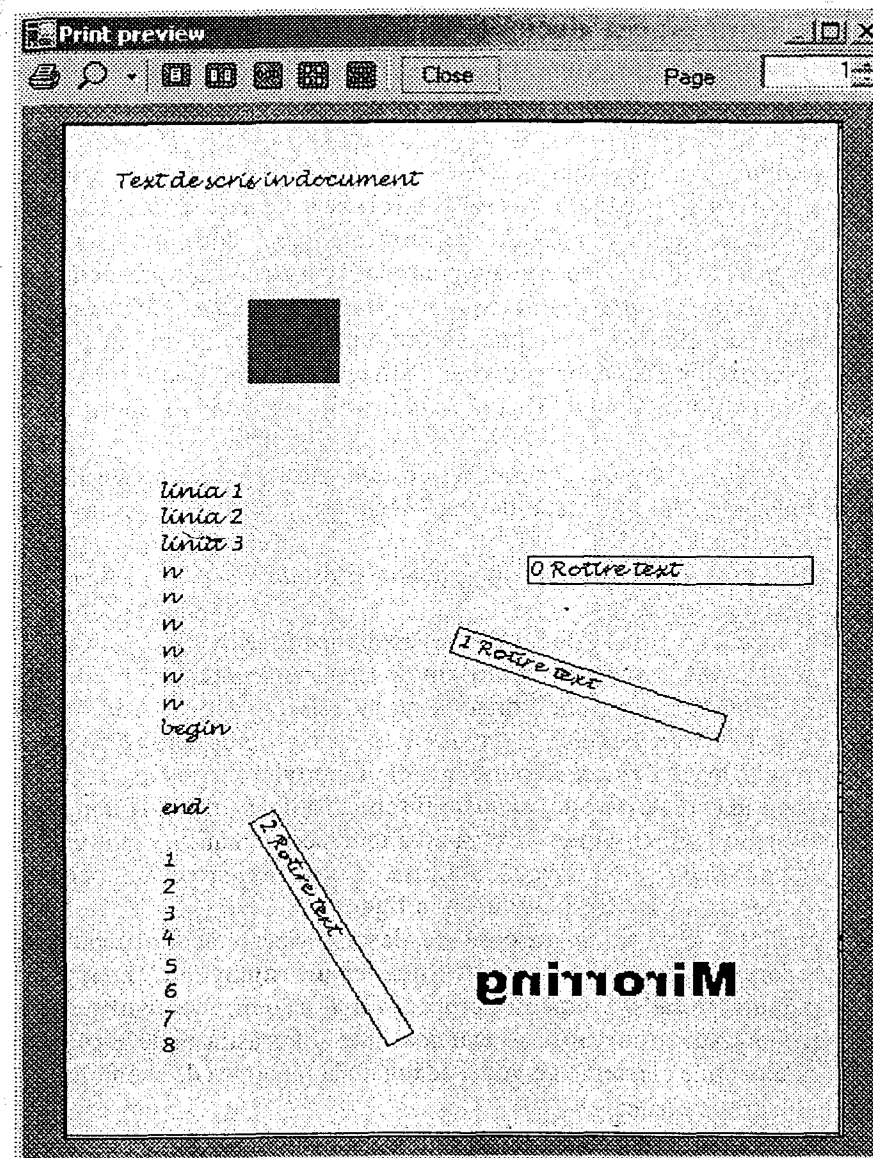

  ev.Graphics.RotateTransform(-60); // revenire la normal

  int cx = (int)ev.Graphics.ClipBounds.Width,
     cy = (int)ev.Graphics.ClipBounds.Height;

  ev.Graphics.TranslateTransform( 100, 100);
  ev.Graphics.ScaleTransform(-1,1);

  fnt = new Font("Arial Black", 40);
  ev.Graphics.DrawString(" Mirorring", fnt, Brushes.Black,
                -650,850, new StringFormat());
```

Secvenţa de mai sus, inserată în funţia **preview_PrintPage**, are drept consecinţă o previzualizare de genul următor:



3. Imprimarea documentelor utilizând generatorul de rapoarte *CrystalReports*

Mediul de programare .NET permite dezvoltarea de aplicaţii software de o mare diversitate. Accesul facil la baze de date de diferite tipuri a permis dezvoltarea de aplicaţii performante şi în domeniul gestiunii volumelor mari de date. Generarea rapoartelor prin mijloacele prezentate în paragrafele anterioare este destul de anevoioasă şi impune un efort mare de programare. Ne referim şi la faptul că imprimarea unor fişe cu un format impus, spre exemplu a unor *fluturaşi* individuali cu date despre salariul lunar, devine o operaţie migăloasă fără un instrument vizual de construire a rapoartelor.

CrystalReports este generatorul de rapoarte inclus în mediul .NET. El funcţionează şi ca o aplicaţie de sine stătatoare şi poate fi invocat şi din aplicaţii de utilizator.

Pentru a genera un raport cu aplicaţia *CrystalReports* din C# se vor parcurge etapele următoare:

1. se crează un proiect de tip *Windows Application*;
2. se adaugă la proiect un nou **Item** de tip *Crystal Report* (în fereastra *Solution Explorer*, click dreapta pe numele proiectului **Add / Add New Item**, se deschide o fereastră de dialog, din panelul *Templates* se selectează ***Crystal Report***; numele raportului se generează, iar în cazul în care se doreşte, el poate fi schimbat doar editând noul nume;
3. se apasă butonul **Open** care va crea un fişier cu numele raportului şi extensia ***rpt*** şi se invocă aplicaţia ***Crytal Reports*** care permite decrierea vizuală a raportului ca în figura 9.1.
4. pe lîngă fereastra în care se va descrie vizual raportul având în vedere structura unui raport: **header-ul raportului**, **header-ul paginii**, **Details**- conţinutul propriu-zis al paginii, **footer-ul paginii** şi **footer-ul raportului**, se mai deschide şi fereastra **Field Explorer** care conţine elemente ce vor fi adăugate la raport, ca de exemplu:
   - sursa de date;
   - câmpuri independente;
   - formule sub forma de câmpuri calculate;
   - câmpuri speciale cum ar fi data şi ora sistem;
   - numărul paginii curente;
   - numărul înregistrării curente etc.

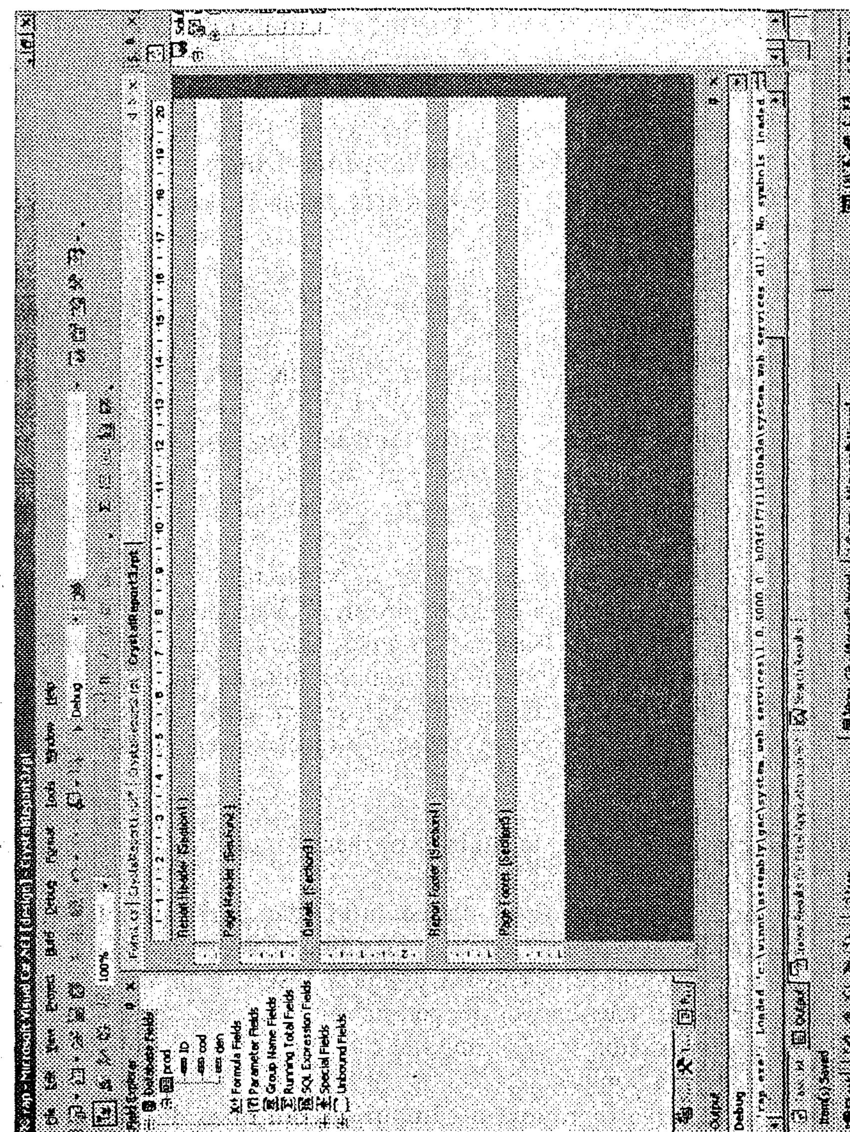


Fig 9.1 Aplicaţia *Crystal Reports*

5. Construirea raportului presupune în primul rînd definirea unor elemente de ansamblu, cum ar fi:
   - setarea imprimantei şi a mărimii paginii (click dreapta pe raport şi se alege opţiunea **Designer / Printer Setup**);
   - stabilirea marginilor (click dreapta pe raport şi se alege opţiunea **Designer / Page Setup**);
   - dacă se doreşte suprimarea unor secţiuni din raport sau adăugarea de noi secţiuni (de exemplu încă o secţiune **Details** care să conţină elementele de imprimat pe verso-ul unei pagini) sau printarea secvenţială a înregistrărilor pînă se completează pagina sau câte o înregistrare pe pagină etc (click dreapta pe raport şi se alege opţiunea **Format Section;** se deschide un dialog cu două panel-uri: cel din stânga conţine secţiunile raportului, iar cel din dreapta, opţiuni referitoare la secţiunea curentă);
   - legarea la o sursă de date se face selectând itemul **Database Fields** din fereastra **Field Explorer**, click dreapta şi se selectează **Add / Remove Database,** se deschide un dialog, în panel-ul **Available Data Sources** se stabileşte sursa de date, iar în cel din dreapta se aleg tabelele din care se vor afişa datele; în itemul **Database Fields** va apare numele tabelei şi câmpurile asociate ei.

6. Definirea şi aranjarea în pagină a elementelor care se vor imprima în raport:
   - **adăugarea de texte statice, linii, câmpuri speciale, sigle, grafice** etc. într-o secţiune se face dînd click dreapta pe secţiunea respectivă şi se selectează opţiunea **Insert**, după care se alege tipul de element dorit pentru a fi inserat şi se editează corespunzător, apoi se mută în cadrul paginii în locul în care se doreşte a fi imprimat;
   - **adăugarea în raport a câmpurilor din baza de date** se face selectând un câmp din fereastra **Field Explorer,** itemul **Database Fields** şi prin *drag and drop* se fixează în secţiunea dorită şi la locul dorit în raport;
   - **adăugarea unui câmp de tip parametru** la raport se face selectînd item-ul **Parameter Fields**, click dreapta şi se selectează **New** după care apare un dialog prin care se specifică în principal numele câmpului şi tipul său; se apasă **OK** şi numele

   câmpului va apare ca subitem al item-ului Parameter Fields; prin *drag and drop* se fixează, în raport, în secţiunea dorită şi la locul dorit;
   - **adăugarea unui câmp de tip formulă,** la raport, se face selectând item-ul **Formula Fields**, click dreapta şi se selectează **New**, se introduce numele formulei după care apare un dialog prin care ea se poate edita vizual ca în figura 9.2.

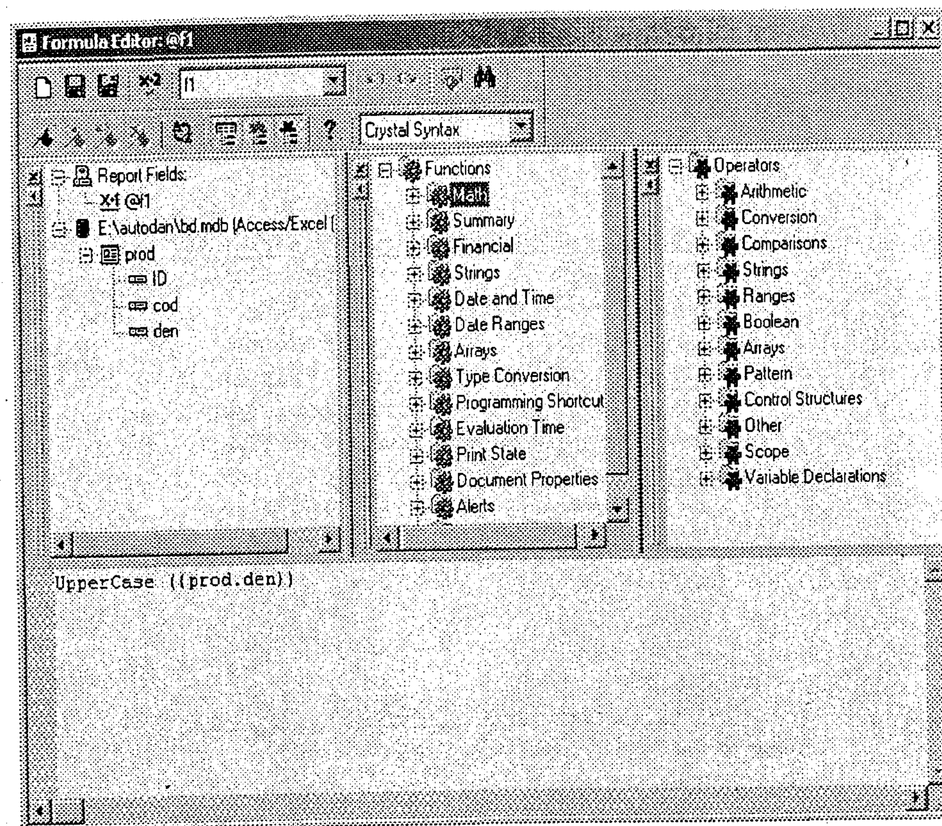


Fig 9.2 Editorul vizual de formule

Formula se compune din operanzi care vor fi preluaţi din panel-ul din stânga, iar prelucrările efectuate asupra lor implică folosirea de operatori (existenţi în panel-ul din dreapta) şi funcţii (în panel-ul din mijloc). De exemplu, construcţia: **UpperCase ({prod.den})** are ca scop conversia şirului denumire (*den*) din tabela *prod* la litere mari. Sintaxa de redactare a formulelor este una specifică soft-ului *Crystal Reports,* dar se poate alege şi

sintaxă tip *Basic*. Inchiderea dialogului se poate face cu salvarea formulei caz în care, dacă ea e scrisă corect sintactic, numele va apare ca subitem în itemul Formula Fields, de unde prin *drag and drop* se fixează în secțiunea dorită şi la locul dorit în raport.

Integrarea raportului într-o aplicație C# presupune utilizarea pe o formă a unui control de tip `CrystalReportViewer`. Legătura dintre control şi raportul propriu-zis se face prin proprietatea `ReportSource` a obiectului de tip `CrystalReportViewer`, adică se introduce numele cu calea către fişierul de tip ***rpt*** care conține raportul.

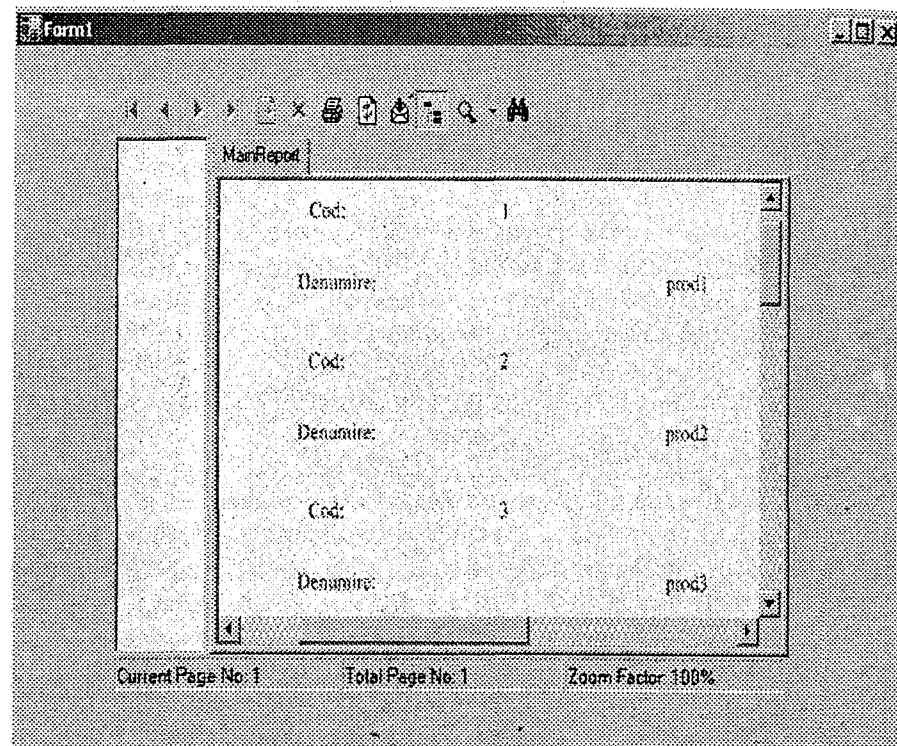


Fig 9.3 Vizualizarea raportului în vederea imprimării

Controlul de tip `CrystalReportViewer` se poate configura după dorința programatorului prin modificarea proprietăților corespunzătoare. El are tools-uri utile pentru

- vizualizarea succesivă a paginilor,
- afişarea primei / ultimei pagini,

- zoom pe document
- conversie document într-un alt format,
- trimiterea documentului spre tipărire la imprimantă etc.

Aceste operații sunt deja implementate şi sunt disponibile prin intermediul butoanelor din bara de instrumente a controlului după cum se poate vedea în figura 9.3.

Din cele prezentate lucrurile sunt relativ simple şi țin mai mult de utilizarea soft-ului *Crystal Reports* decât de programarea în C#, acest lucru se datorează în principal faptului că am lucrat cu o sursă de date deja constituită.

Dacă dorim să parametrizăm un raport iar datele nu le vom prelua dintr-o sursă de date, ci le vom prelua din controale (variabile) existente într-o aplicație C#, atunci comunicarea între aplicație şi raport nu se va mai putea face numai vizual. Pentru parametrizarea raportului se vor folosi, în raport, obiecte de tip **Parameter Fields**.

Pentru exemplificare vom contrui o aplicație (un proiect de tip Windows Application), pe forma principală `Form1` vom cere introducerea următoarelor date: nume, prenume, numărul de ore şi salariul orar şi vom pune un buton `Raport` la apăşarea căruia se va vizualiza un raport.

In figura 9.4 se poate observa forma principală a aplicației.

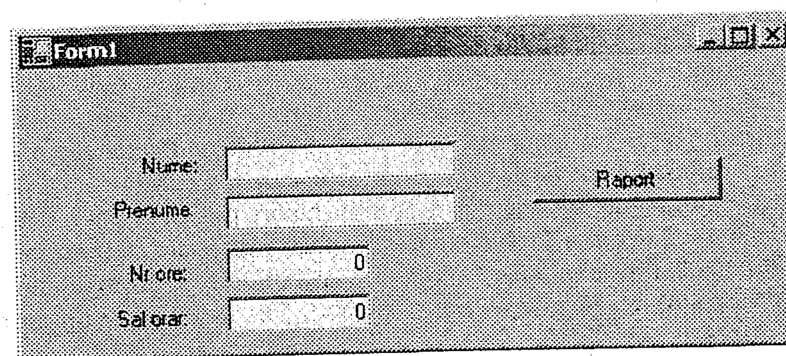


Fig. 9.4 Forma de introducere a datelor

Raportul se va construi cu *Crystal Reports* şi are forma prezentată în figura 9.5.

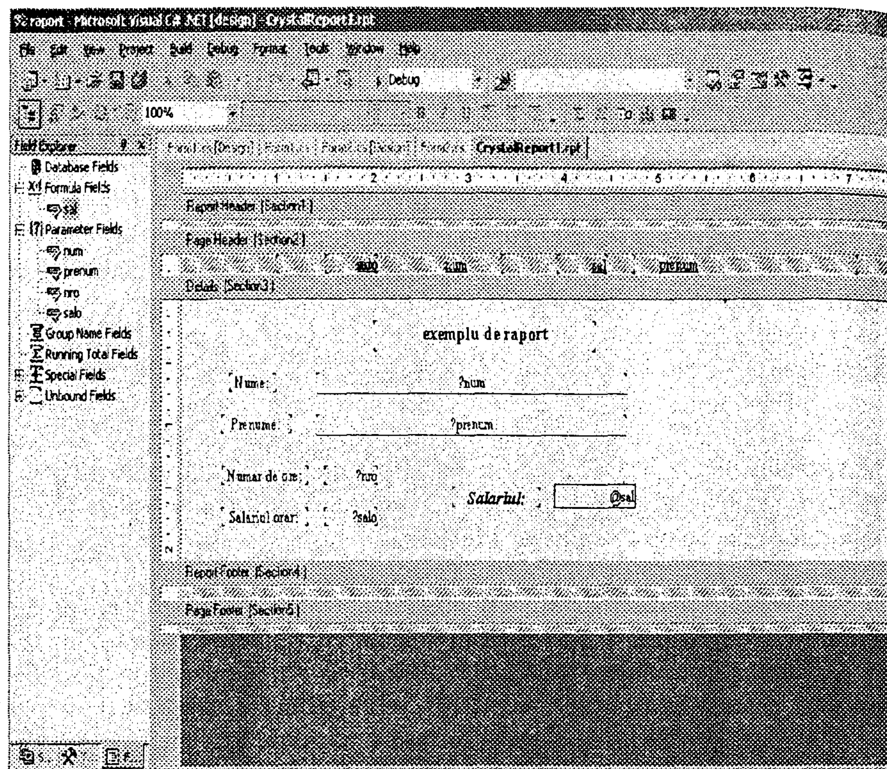


Fig 9.5 Machetă de raport

Structura fixă a raportului a fost construită folosind obiecte de tip **TextObject**, partea variabilă este dată de membrii de tip **Parameter Fields:** ***num*** – pentru nume, ***prenum*** – pentru prenume, ***nro*** – pentru număr de ore şi ***salo*** – pentru salariul orar, care apar în raport precedaţi de **?** şi de un membru de tip **Formula Fields;** spre exemplu, ***sal*** care are ca scop calculul salariului după formula: *nro* * *salo*, are în editorul de formule sintaxa: **{?nro}** * **{?salo}**. Se poate observa în figura 9.5 că un câmp de tip formulă este precedat de @.

Valorile parametrilor se vor prelua din forma principală şi se vor afişa în raportul construit anterior.

Prin construirea raportului se generează fişierul cu nume implicit *CrystalReport1.rpt* care conţine descrierea raportului şi în plus la proiect apar încă două obiecte: `CrystalReport1` şi `CacheCrystalReport1`.

Pentru a vizualiza raportul vom adăuga la proiect o nouă formă (`Form2`) pe care vom adăuga un control de vizualizare a raportului (de tip **`crystalReportViewer`**, variabila fiind **`crystalReportViewer1`**) şi vom stabili legătura către raportul de vizualizat (proprietatea **`ReportSource`**).

In forma principală, la apăsarea pe butonul **`Raport`**, se va vizualiza fereastra `Form2`:

```
private void button1_Click(object sender, System.EventArgs e)
      {
            Form2 fr=new Form2(this);
            fr.ShowDialog();
      }
```

Accesarea membrilor clasei `Form1` din clasa `Form2` se va face trimiţând constructorului clasei `Form2` o referinţă la clasa `Form1`. Constructorul clasei `Form2` preia referinţa într-o variabilă (**`fp`**) şi prin intermediul ei se trimit raportului valorile introduse în forma `Form1` pentru a fi tipărite:

```
public Form2(Form1 k)
{
      InitializeComponent();  fp=k;
// creare obiect de tip parametru
      ParameterDiscreteValue num =
                                new ParameterDiscreteValue();
// preluarea valorii parametrului, din controlul textBox1
      num.Value = fp.textBox1.Text;
// asocierea acestui parametru ca valoare curenta a primului
// cimp parametru din raport (?num)
      crystalReportViewer1.ParameterFieldInfo[0].CurrentValue
                                              s.Add(num);

// acelasi lucru pentru ceilalti parametri
      ParameterDiscreteValue pren =
                                new ParameterDiscreteValue();
      pren.Value = fp.textBox2.Text;
      crystalReportViewer1.ParameterFieldInfo[1].CurrentValue
                                              s.Add(pren);

      ParameterDiscreteValue no =
                                new ParameterDiscreteValue();
      no.Value  = fp.textBox3.Text;
      crystalReportViewer1.ParameterFieldInfo[2].CurrentValue
                                              s.Add(no);

      ParameterDiscreteValue so =
                                new ParameterDiscreteValue();
      so.Value  = fp.textBox4.Text;
      crystalReportViewer1.ParameterFieldInfo[3].CurrentValue
                                              s.Add(so);

}
```

După executarea constructorului, în fereastră se va afişa raportul cu valorile curente, ca în figura 9.6.

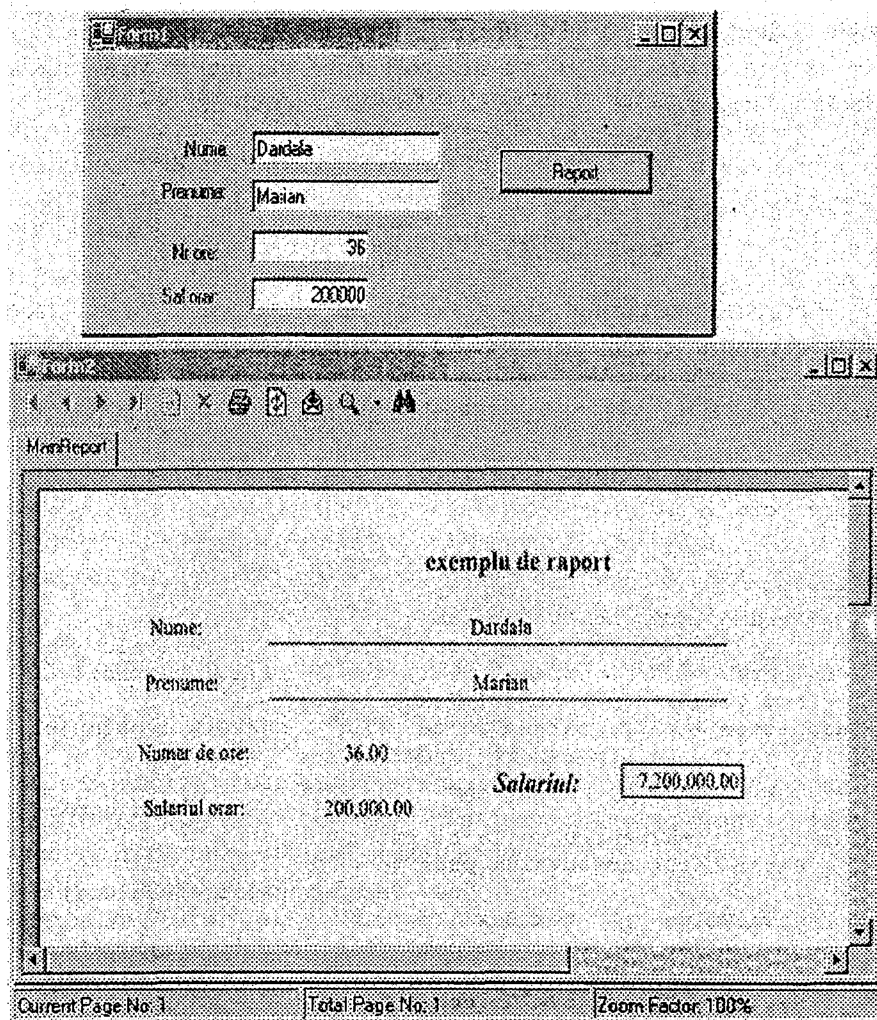


Fig 9.6 Afişarea raportului cu date preluate dintr-o formă

VIZUALIZĂRI SPLITATE. ELEMENTE DE GRAFICĂ

1. **Docarea şi ancorarea controalelor. Vizualizări splitate. Panel-uri**
2. **Elemente de grafică**

1. Docarea şi ancorarea controalelor. Vizualizări splitate. Panel-uri

Controalele mai dispun de o proprietate interesantă numită `Dock`; ea stabileşte de care margine a containerului să se "lipească" unele controale : `Left`, `Right`, `Top`, `Bottom` sau `Fill`, când trebuie să umple întreg containerul (uzual, forma).

Proprietatea `Dock` nu trebuie confundată cu proprietatea `Anchor`, care stabileşte reperul indicat de data aceasta prin două margini (`Left - Top`, `Right-Top` etc.) faţă de care controlul va păstra aceeaşi distanţă (iniţială) în momentul în care containerul se redimensionează (spre exemplu, la momentul execuţiei, utilizatorul trage de colţul dreapta-jos al formei, mărind-o).

Din considerente estetice, docarea nu se aplică unor controale gen `Button` sau `Label`, dar pare justificată pentru controale de tip `TextBox`, `ListBox` etc., asigurând legarea lor de o margine şi extinderea lor la dimensiunea întregii margini.

> Dacă un control nu este vizibil pe formă, sau selectarea lui este dificilă datorită dimensiunii mici sau plasării în spatele altui control, se poate folosi fereastra Properties, care are în partea superioară un ComboBox, în care apar toate controalele formei, putând fi astfel uşor selectate.

Controlul Splitter

În ToolBox există un control interesant numit **Splitter**, care adus pe o formă cu două controale, se poziţionează automat între ele. El apare ca o linie punctată, iar pentru a-i mări vizibilitatea îl putem colora (proprietatea `BackColor`) într-o culoare aprinsă.

La momentul execuției, dacă se trage de splitter, controalele din jurul său se redimensionează automat, unul mărindu-se, altul micşorându-se.

Splitter-ul lucrează diferit în funcție de proprietatea sa `Dock`; implicit ea este `Left`. Dacă pe formă se găseşte deja un control docat `Left`, splitter-ul se poziționează în continuarea controlului, respectiv pe marginea dreaptă a acestuia. Mişcarea splitului la momentul rulării, determină redimensionarea controlului de care este docat. Se poate deduce uşor comportamentul dacă proprietatea `Dock` a splitter-ului se alege `Right`, `Top` sau `Bottom`; putem aşadar să scindăm o fereastră pe orizontală şi pe verticală, obținând astfel mai multe **zone funcționale**.

O întrebare firească este ce se întâmplă când avem mai multe controale pe formă sau sunt controale « nedocabile », din considerente estetice.

Soluția o oferă controlul **Panel**, din ToolBox. El are două funcțiuni majore :

- oferă suportul pentru **desenare** (*canvas*), dispunând de un context grafic ce poate fi extras deoarece panel-ul recunoaşte evenimentul `Paint`;
- acționează drept **container** (control care ține alte controale), iar lipit de un splitter, se mută luând cu el toate controalele pe care le conține. A doua funcțiune poate fi suplinită şi printr-un control GroupBox, dar acesta oferă mai puține facilități.

2. Elemente de grafică

Suportul pentru reprezentările grafice este oferit în Visual C# de biblioteca de clase .NET prin intermediul clasei **Graphics**. Ea conține toate funcțiile de trasare grafică, precum şi informațiile despre zona client, funcții pentru principalele transformări matematice, cum ar fi curbe Bezier, rotația, translația, scalarea etc.

Clasa **Graphics** este *sealed* şi deci nu putem moşteni din ea prin derivarea unei clase noi.
Nu are constructor, obiectele ei fiind obținute pe căi indirecte, dintre care cele mai frecvente sunt prezentate în continuare.

1. Evenimentul **`Paint`** are argumentul **`PaintEventArgs`**, care are la rândul lui proprietatea **Graphics** ce referă un obiect **Graphics** asociat suprafeței grafice a obiectului ce a inițiat **`Paint`**-ul.
2. Apelând funcția **`CreateGraphics()`** specifică unui control.


```
private void btn_Click(object sender, System.EventArgs e)
{
        Graphics g = button1.CreateGraphics();
        g.DrawEllipse(new Pen(Color.Red,2), 10,10,20,20);
}
```

Funcția de mai sus trasează o elipsă chiar pe suprafața unui buton.
Contează unde pun codul pentru trasare; pus în constructorul clasei nu se vede, căci forma se retrasează. Pus pe click buton ca mai sus, dacă forma are **`this.ResizeRedraw = true;`** şi s-a acoperit prin redimensionare şi butonul, când acesta reapare nu mai are desenul şi trebuie din nou dat click.

3. Apelând **`Graphics.FromHwnd(this.Handle)`** şi furnizându-i handler-ul ferestrei formei sau controlului repectiv.
4. Apelând metoda statică **`Graphics.FromImage(bmp)`**, care pornind de la o imagine (declarată sub forma **`Bitmap bmp;`**) returnează un obiect grafic asociat acesteia.

Exercițiu

Să se exemplifice elementele de bază ale unei reprezentări grafice, folosind contextul grafic al formei principale a aplicației, obținut din blocul de parametri ai funcției de tratare a evenimentului Paint al formei.

Rezolvare

```
protected override void
            OnPaint(System.Windows.Forms.PaintEventArgs pe)
{
  base.OnPaint(pe); // eventual, invocarea metodei din baza

  this.ResizeRedraw = true;
  // invocare Paint la orice redimensionare
  // a se testa si cu proprietatea pe false

  //obtinere obiect grafic din parametrii lui Paint
  Graphics g = pe.Graphics;

  // extragerea dreptunghiului in care se poate desena
  Rectangle zonaClient=pe.ClipRectangle;
  // puteam folosi si proprietatea ClientRectangle a formei
  // Rectangle zonaClient = ClientRectangle;

  Brush fundal=new SolidBrush(Color.White);
  g.FillRectangle(fundal,zonaClient);
                    //coloreaza alb zonaClient
```

```
zonaClient.X+=5; zonaClient.Y+=5;        // margini
zonaClient.Height-=10; zonaClient.Width-=10;
                  // dreptunghi de vizualizare

Pen creionRosu = new Pen(Color.Red,2);  // rosu, grosime 2
g.DrawRectangle(creionRosu, zonaClient);

// coordonate calculate functie de dimensiune fereastra;
// figurile se redimensioneaza odata cu fereastra

Point stgSus = new Point(zonaClient.X,zonaClient.Y);
Point drtJos = new Point( zonaClient.X+zonaClient.Width,
                          zonaClient.Y+zonaClient.Height );

g.DrawLine(creionRosu,stgSus,drtJos);
SolidBrush pnsGalbena = new SolidBrush(Color.Yellow);

PointF p = new PointF(  zonaClient.X+zonaClient.Width/4,
                        zonaClient.Y+zonaClient.Height/3 );

SizeF s =new SizeF(zonaClient.Width/2,zonaClient.Height/3);

RectangleF drept = new RectangleF(p,s);

g.FillEllipse(pnsGalbena,drept);  // figuri pline

// lucru in coordonate absolute;
// figurile raman pe loc la redimensionare fereastra
SizeF size=new SizeF(100, 150);
PointF point=new PointF(20,40);
RectangleF rec =new RectangleF(point,size);

Pen pa=new Pen(Color.Blue, 3); // albastru grosime 3
g.DrawEllipse(pa,rec);

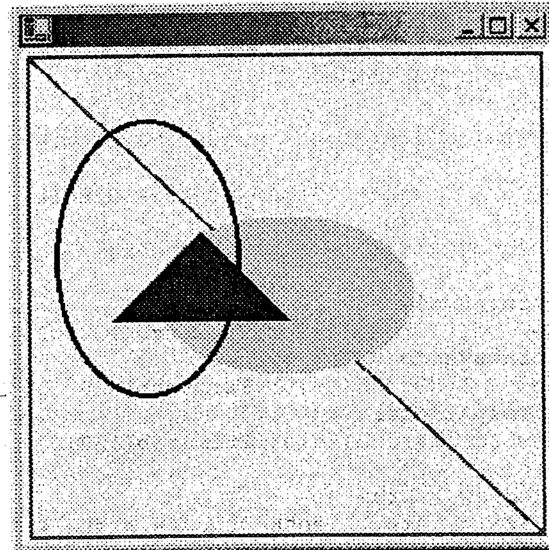
Point p1=new Point(100,100),
                p2 = new Point(50,150), p3=new Point();
p3.X=150; p3.Y=150;
Point []vp=new Point[3]{p1,p2,p3};

SolidBrush pnsAlbastra = new SolidBrush(Color.Blue);
g.FillPolygon(pnsAlbastra,vp);
}
```

Se observă că există versiuni duble ale unor clase, în funcție de tipul coordonatelor folosite, `int` sau `float`: **`Point`**, **`PointF`**, **`Rectangle`**, **`RectangleF`**, **`Size`**, **`SizeF`**.

Atunci cînd se desenează text, programatorul trebuie să calculeze coordonatele de la care începe scrierea, având grijă să avanseze, linie cu linie. O modalitate de calcul al coordonatelor se bazează pe extragerea dimensiunii textului scris (lungime și înălțime) în funcție de numărul de caractere și de fontul cu care este scris textul:

```
SizeF lg_lat  grfx.MeasureString(sir,font);
```



Dacă interesează aceste coordonate doar pentru centrarea textului, o variantă mult mai comodă uzează de facilitățile unui obiect de tip `StringFormat`, care preia toată sarcina alinierii:

```
StringFormat sf; sf.Alignment = StringAlignment.Center;
```

Programatorul nu trebuie decât să aleagă din enumerarea `StringAlignment`, pe cea dorită.

`ResizeDraw = true;` etse o proprietate a formei care solicită ca la fiecare redimensionare să se apeleze și funcția `Invalidate()`, adică funcția de retrasare a ferestrei prin invocarea metodei **Paint**.

Alpha blending este o tehnică de amestecare a culorilor, astfel încât la suprapunerea a două desene este creată senzația de transparență și de mixare a culorilor, observându-se astfel intersecția obiectelor. Tehnica alpha blending este disponibilă prin intermediul culorii pensulelor; astfel, la

crearea culorii pensulelor se indică drept prim parametru un factor de mixare a culorilor, care ia valori tot între 0 şi 255:

```
SolidBrush trnsRedBrush =
        new SolidBrush(Color.FromArgb(120, 255, 0, 0));
```

Funcţia de mai jos declanşată la un click de mouse pe formă, calculează centrele şi razele a trei cercuri astfel încât acestea să aibă tot timpul zone suprapuse; în felul acesta, colorarea cu tehnica alpha blending este mai bine pusă în evidenţă, mai ales că ea se aplică între culorile elementare: roşu, verde şi albastru.

```
private void Form1_Click(object sender, System.EventArgs e)
{
 Graphics     g = Graphics.FromHwnd(this.Handle);

 // Pensule transparente cu alpha blending

 SolidBrush trnspRosie =
           new SolidBrush(Color.FromArgb(120, 255, 0, 0));
 SolidBrush trnspVerde =
           new SolidBrush(Color.FromArgb(120, 0, 255, 0));
 SolidBrush trnspAlbastra =
           new SolidBrush(Color.FromArgb(120, 0, 0, 255));

 this.ResizeRedraw = true;
 float   triBaza = this.ClientRectangle.Width/4;
 float   triInaltime =(float)Math.Sqrt(3)*triBaza/(float)2;

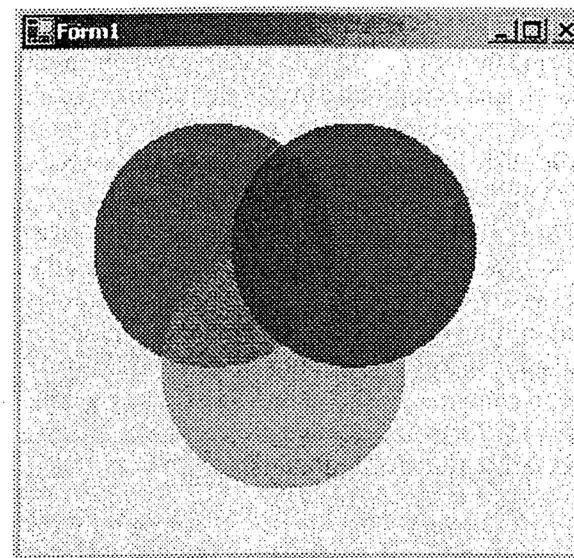
 float   x1 = 40;      float   y1 = 40;

 g.FillEllipse(trnspRosie, x1, y1, 2*triInaltime,
                                   2*triInaltime);

 g.FillEllipse(trnspVerde, x1 + triBaza/2, y1 + triInaltime,
                           2*triInaltime, 2*triInaltime);

 g.FillEllipse(trnspAlbastra, x1 + triBaza, y1,
                           2*triInaltime, 2*triInaltime);

}
```

Exerciţiu

Să se scrie o funcţie pentru **reprezentarea grafică a unei curbe** oarecare prin trasarea ei, **pixel cu pixel.**

Rutina de trasare grafică porneşte de la conceptele de fereastră şi vizor. Scalarea imaginii se va face astfel încât maximul şi minimul funcţiei să poată fi reprezentate în vizor.

Testul de generalitate se va face apelând rutina pentru o parabolă şi respectiv, pentru o sinusoidă.

Rezolvare

Elemente pregătitoare:

- O **structură de date** care grupează informaţiile preluate din fereastra utilizator şi care vor fi transmise clasei responsabile cu reprezentarea grafică.
- Un `delegate` ce conţine funcţia de reprezentat grafic.

De data aceasta, programul este structurat pe **două clase,** pornind de la ideea că nu trebuie totul îngrămădit în clasa Form1. Ea trebuie să concentreze doar elementele ce ţin de interfaţa grafică cu utilizatorul: preluarea datelor sau parametrilor de reprezentare grafică, butoane de control sau pentru alegerea tipului de reprezentare grafică. Misiunea ei se

încheie după ce a împachetat datele şi parametrii şi le-a transferat funcţiei de trasare grafică, prin lansarea metodei de tratare a evenimentului `Paint`.

Deşi există o clasă **Graphics** care conţine elementele de bază, inclusiv metodele, necesare reprezentării grafice, s-a preferat definirea şi a unei clase de utilizator, **grafic**, care să înglobeze culori, pensule, creioane, funcţii de reprezentare grafică mai complicate, create de programator.

În moment tratării evenimentului Paint se instanţiază obiectul grafic de utilizator, i se pasează contextul Graphics şi i se predă controlul prin apelul **`g.Arata(e)`**.

La rândul ei, funcţia definitivează contextul grafic completându-l cu elemente care nu au fost date din exterior de către utilizator.

Pentru reprezentarea grafică propriu-zisă, clasa **grafic** preia datele şi contextul grafic al formei, efectuează calculele matematice, setează parametrii grafici doriţi şi trasează graficul.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace graf_pixel
{
public class Form1 : System.Windows.Forms.Form
 {
  private System.ComponentModel.Container components = null;
  public Form1()
  {
      InitializeComponent();
  }

 protected override void Dispose( bool disposing )
  {
      if( disposing )
      {
            if (components != null)
            {
                  components.Dispose();
            }
      }
      base.Dispose( disposing );
  }

#region Windows Form Designer generated code
```



```
private void InitializeComponent()
 {
  this.panel1 = new System.Windows.Forms.Panel();
  this.mes = new System.Windows.Forms.TextBox();
  this.lbC = new System.Windows.Forms.Label();
  this.tbC = new System.Windows.Forms.TextBox();
  this.label4 = new System.Windows.Forms.Label();
  this.tbB = new System.Windows.Forms.TextBox();
  this.label3 = new System.Windows.Forms.Label();
  this.tbA = new System.Windows.Forms.TextBox();
  this.label2 = new System.Windows.Forms.Label();
  this.tb = new System.Windows.Forms.TextBox();
  this.label1 = new System.Windows.Forms.Label();
  this.ta = new System.Windows.Forms.TextBox();
  this.groupBox1 = new System.Windows.Forms.GroupBox();
  this.radioButton2 = new System.Windows.Forms.RadioButton();
  this.radioButton1 = new System.Windows.Forms.RadioButton();
  this.panel2 = new System.Windows.Forms.Panel();
  this.panel1.SuspendLayout();
  this.groupBox1.SuspendLayout();
  this.SuspendLayout();
      //
      // panel1- interfata cu utilizatorul
      //
  this.panel1.Controls.Add(this.mes);
  this.panel1.Controls.Add(this.lbC);
  this.panel1.Controls.Add(this.tbC);
  this.panel1.Controls.Add(this.label4);
  this.panel1.Controls.Add(this.tbB);
  this.panel1.Controls.Add(this.label3);
  this.panel1.Controls.Add(this.tbA);
  this.panel1.Controls.Add(this.label2);
  this.panel1.Controls.Add(this.tb);
  this.panel1.Controls.Add(this.label1);
  this.panel1.Controls.Add(this.ta);
  this.panel1.Controls.Add(this.groupBox1);
  this.panel1.Dock = System.Windows.Forms.DockStyle.Left;
  this.panel1.Location = new System.Drawing.Point(0, 0);
  this.panel1.Name = "panel1";
  this.panel1.Size = new System.Drawing.Size(176, 293);
  this.panel1.TabIndex = 0;
      //
      // mes
      //
  this.mes.Location = new System.Drawing.Point(0, 120);
  this.mes.Name = "mes";
  this.mes.Size = new System.Drawing.Size(160, 20);
  this.mes.TabIndex = 11;
  this.mes.Text = "";
      //
```

```
    //  lbC
    //
this.lbC.Location = new System.Drawing.Point(16, 88);
this.lbC.Name = "lbC";
this.lbC.Size = new System.Drawing.Size(9, 16);
this.lbC.TabIndex = 10;
this.lbC.Text = "C";
    //
    // tbC
    //
this.tbC.Location = new System.Drawing.Point(32, 88);
this.tbC.Name = "tbC";
this.tbC.Size = new System.Drawing.Size(48, 20);
this.tbC.TabIndex = 9;
this.tbC.Text = "1";
this.tbC.Leave += new System.EventHandler(this.Pleaca);
    //
    // label4
    //
this.label4.Location = new System.Drawing.Point(16, 56);
this.label4.Name = "label4";
this.label4.Size = new System.Drawing.Size(9, 16);
this.label4.TabIndex = 8;
this.label4.Text = "B";
    //
    // tbB
    //
this.tbB.Location = new System.Drawing.Point(32, 56);
this.tbB.Name = "tbB";
this.tbB.Size = new System.Drawing.Size(48, 20);
this.tbB.TabIndex = 7;
this.tbB.Text = "1";
this.tbB.Leave += new System.EventHandler(this.Pleaca);
    //
    // label3
    //
this.label3.Location = new System.Drawing.Point(16, 24);
this.label3.Name = "label3";
this.label3.Size = new System.Drawing.Size(9, 16);
this.label3.TabIndex = 6;
this.label3.Text = "A";
    //
    // tbA
    //
this.tbA.Location = new System.Drawing.Point(32, 24);
this.tbA.Name = "tbA";
this.tbA.Size = new System.Drawing.Size(48, 20);
this.tbA.TabIndex = 5;
this.tbA.Text = "1";
this.tbA.Leave += new System.EventHandler(this.Pleaca);
```



```
    //
    // label2
    //
this.label2.Font = new System.Drawing.Font
   ("Microsoft Sans Serif", 10F,
    System.Drawing.FontStyle.Bold,
    System.Drawing.GraphicsUnit.Point,((System.Byte)(0))
    );
this.label2.Location = new System.Drawing.Point(128, 160);
this.label2.Name = "label2";
this.label2.Size = new System.Drawing.Size(16, 23);
this.label2.TabIndex = 4;
this.label2.Text = "]";
    //
    // tb- dreapta intervalului de trasare
    //
this.tb.Location = new System.Drawing.Point(96, 160);
this.tb.Name = "tb";
this.tb.Size = new System.Drawing.Size(24, 20);
this.tb.TabIndex = 3;
this.tb.Text = "10";
    //
    // label1
    //
this.label1.Font = new System.Drawing.Font
    ("Microsoft Sans Serif", 10F,
    System.Drawing.FontStyle.Bold,
    System.Drawing.GraphicsUnit.Point, ((System.Byte)(0))
    );
this.label1.Location = new System.Drawing.Point(32, 160);
this.label1.Name = "label1";
this.label1.Size = new System.Drawing.Size(16, 23);
this.label1.TabIndex = 2;
this.label1.Text = "[";
    //
    // ta - stanga intervalului de trasare
    //
this.ta.Location = new System.Drawing.Point(56, 160);
this.ta.Name = "ta";
this.ta.Size = new System.Drawing.Size(24, 20);
this.ta.TabIndex = 1;
this.ta.Text = "-10";
    //
    // groupBox1 - functiile de trasat
    //
this.groupBox1.Controls.Add(this.radioButton2);
this.groupBox1.Controls.Add(this.radioButton1);
this.groupBox1.Location =
                        new System.Drawing.Point(24, 192);
this.groupBox1.Name = "groupBox1";
```

```
this.groupBox1.Size = new System.Drawing.Size(136, 88);
this.groupBox1.TabIndex = 0;
this.groupBox1.TabStop = false;
this.groupBox1.Text = "functia";
    //
    // radioButton2 - optiune sinus
    //
this.radioButton2.Location =
          new System.Drawing.Point(16, 48);
this.radioButton2.Name = "radioButton2";
this.radioButton2.TabIndex = 1;
this.radioButton2.Text = "sin";
this.radioButton2.CheckedChanged += new System.EventHandler
                    (this.radioButton2_CheckedChanged);
    //
    // radioButton1 - optiune parabola
    //
this.radioButton1.Checked = true;
this.radioButton1.Location =
          new System.Drawing.Point(16, 16);
this.radioButton1.Name = "radioButton1";
this.radioButton1.TabIndex = 0;
this.radioButton1.TabStop = true;
this.radioButton1.Text = "parabola";
this.radioButton1.CheckedChanged += new System.EventHandler
                    (this.radioButton1_CheckedChanged);
    //
    // panel2 - reprezentarea grafica
    //
this.panel2.BackColor =
    System.Drawing.Color.FromArgb(((System.Byte)(255)),
           ((System.Byte)(255)), ((System.Byte)(192)));
this.panel2.Dock = System.Windows.Forms.DockStyle.Fill;
this.panel2.Location = new System.Drawing.Point(176, 0);
this.panel2.Name = "panel2";
this.panel2.Size = new System.Drawing.Size(280, 293);
this.panel2.TabIndex = 1;
this.panel2.SizeChanged +=
          new System.EventHandler(this.panel2_SizeChanged);
this.panel2.Paint +=
          new System.Windows.Forms.PaintEventHandler
                              (this.panel2_Paint);
    //
    // Form1
    //
this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
this.ClientSize = new System.Drawing.Size(456, 293);
this.Controls.Add(this.panel2);
this.Controls.Add(this.panel1);
this.Name = "Form1";
```



```
  this.Text = "Grafic prin pixeli";
  this.panel1.ResumeLayout(false);
  this.groupBox1.ResumeLayout(false);
  this.ResumeLayout(false);
 }
#endregion

[STAThread]
static void Main()
{
      Application.Run(new Form1());
}
private System.Windows.Forms.Panel panel1;
private System.Windows.Forms.Panel panel2;
private System.Windows.Forms.GroupBox groupBox1;
private System.Windows.Forms.RadioButton radioButton1;
private System.Windows.Forms.RadioButton radioButton2;
private System.Windows.Forms.Label label1;
private System.Windows.Forms.Label label2;
private System.Windows.Forms.TextBox ta;
private System.Windows.Forms.TextBox tb;
private System.Windows.Forms.TextBox tbA;
private System.Windows.Forms.Label label3;
private System.Windows.Forms.Label label4;
private System.Windows.Forms.TextBox tbB;
private System.Windows.Forms.Label lbC;
private System.Windows.Forms.TextBox tbC;
private System.Windows.Forms.TextBox mes;
public date dat; //datele de transmis

private void panel2_Paint
      (object sender, System.Windows.Forms.PaintEventArgs e)
 {
   dat=new date();
   if(radioButton1.Checked==true)dat.tip_fct=1;
       else dat.tip_fct=2;
   try
    {
      dat.a= Convert.ToDouble(ta.Text);
      dat.b= Convert.ToDouble(tb.Text);
    }
   catch { dat.a= -10; dat.b=10;}
        // valori implicite pentru intervalul [a, b]

   try
      {
            dat.A= Convert.ToDouble(tbA.Text);
            dat.B= Convert.ToDouble(tbB.Text);
            dat.C= Convert.ToDouble(tbC.Text);
      }
```

```
   catch { dat.A= 1; dat.B=1;dat.C=1; }
       // valori implicite pentru A,B,C
       //dat.tip_fct=1;
   grafic g = new grafic(dat); // context grafic de utilizator
   g.Arata(e); // cere vizualizarea lui si-i preda controlul
  }

   // retrasare grafic la redimensionare panel
  private void panel2_SizeChanged
                            (object sender, System.EventArgs e)
      {     Invalidate(true);        }

 private void radioButton1_CheckedChanged
                            (object sender, System.EventArgs e)
 {
   if(radioButton1.Checked)
            { tbA.Enabled=tbB.Enabled= tbC.Enabled= true;}
   else     { tbA.Enabled=tbB.Enabled= tbC.Enabled= false;}
  }

 private void Pleaca(object sender, System.EventArgs e)
 {//retrasare grafic la parasire camp de editare param a,b,c
      panel2.Invalidate();
 }

 private void radioButton2_CheckedChanged
                           (object sender, System.EventArgs e)
 {
   if(radioButton1.Checked) {dat.a*=Math.PI; dat.b*=Math.PI;}
   else {dat.a/=Math.PI; dat.b/=Math.PI;}
   panel2.Invalidate();
  }
 }// end class Form1

 public struct date
 {
  public int tip_fct;     // parabola sau sinus
  public double a,b;      // interval [a,b]
  public double A,B,C;    // parametri parabola
 }

 delegate double fct(double x);
 // pointer la functia de reprezentat grafic

 public class grafic
 {
  protected Pen pen = new Pen(Color.Red,5);
  protected SolidBrush  brush;
  Color [] culori;
  date dat;
```



```
protected Rectangle vizor; // fereastra de vizualizare

public grafic( date d)
          // constructor context grafic de utilizator
{
     pen = new Pen(Color.Black);
     brush= new SolidBrush(Color.Red);
     dat=d;
}

public void Arata(PaintEventArgs e)
                         // preluare context de la Paint
{
 vizor = e.ClipRectangle; // zona de reprezentare grafica
 vizor.X+=10;             vizor.Y+=10;
 vizor.Height-=20;        vizor.Width-=20; // lasa margini
 e.Graphics.DrawRectangle(pen,vizor);
 fct pf;
 if(dat.tip_fct==1)       pf= new fct(f);
          else            pf= new fct(Math.Sin);
 double yMin, yMax;      MinMax( out yMin, out yMax);
          // MessageBox.Show("Are "+yMin+" si "+yMax);
 RectangleF w = new RectangleF(
          (float)dat.a,            (float)yMin,
          (float)(dat.b-dat.a), float)(yMax-yMin)     );

 graf(pf,w,e);
}


double f( double x)
     {     return dat.A * x*x+ dat.B *x+ dat.C ;      }

double sinus(double x) { return Math.Sin(x); }

void graf(fct pf, RectangleF w, PaintEventArgs e )
 {
  double a1,b1=0,a2,b2=0,x,y,div; int oldx, oldy;
  int vl=vizor.Left,    vt=vizor.Top,
      vb=vizor.Bottom, vr=vizor.Right;

  double wl=w.Left,wr=w.Right, wt=w.Top,wb=w.Bottom;
  string mes="Fereastra "+wl+" si"+wr+ " cu "+wt+" si" +wb;
      //MessageBox.Show(mes);
  int xe,ye;
  try
   {
      a1=(vl-vr)/(wl-wr); b1= (wl*vr-wr*vl)/(wl-wr);
      a2=(vt-vb)/(wb-wt); b2= (wb*vb-wt*vt)/(wb-wt);
      div=(wr-wl)/(vr-vl);
```

```
        xe=(int)(a1*wl+b1);y= pf(wl); ye = (int)(a2*y+b2);

        for(x=wl; x<wr; x+=div)
         {
              oldx=xe;oldy=ye;
              xe=(int)(a1*x+b1); y=pf(x); ye = (int)(a2*y+b2);
              e.Graphics.DrawLine(pen,oldx,oldy,xe,ye);
         }
      }

    catch { MessageBox.Show("Eroare in datele de intrare");
   }

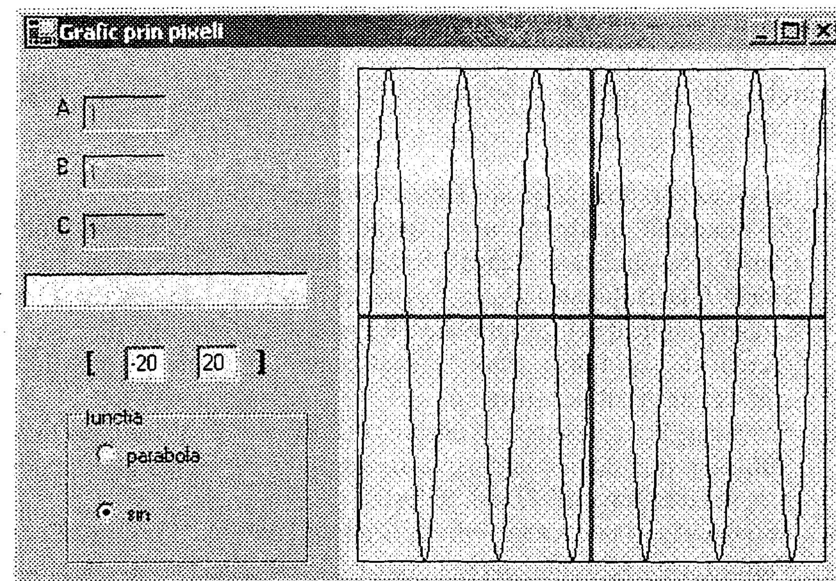
   xe=(int)b1; ye=(int)b2;
   pen=new Pen(brush,3);
   e.Graphics.DrawLine(pen,vl,ye, vr,ye); // axa Ox
   e.Graphics.DrawLine(pen,xe,vt, xe,vb); // axa Oy
  }

void MinMax( out double yMin, out double yMax)
{
  yMin=100.0; yMax=100.0;
  switch(dat.tip_fct)
   {
     case 1:
      {
        double yextr,ys,yd,xextr;
        xextr = -dat.B/2/dat.A; yextr = f(xextr);
        ys = f(dat.a);   yd = f(dat.b);

        if (xextr >= dat.a && xextr <= dat.b)
                                 // extrem de derivata
         if (dat.A>0) // are minim
           { yMin = yextr; yMax = ys>yd ? ys:yd;}
         else {   yMax=yextr; yMin = ys<yd ? ys:yd;   }// max

        else      // doar extreme de frontiera
         { yMax = ys > yd ? ys:yd; yMin = ys < yd ? ys:yd ;}
        break;
      }

     case 2:
      {  yMin = -1.0;  yMax = 1.0; }; break;   }
  }  // end switch
}
}// end user graphic context class
```

Se observă că în funcția de trasare propriu-zisă, deoarece pe contextul grafic nu există o funcție `PutPixel`, s-a folosit funcția `DrawLine`.

Limitarea reprezentării unei curbe prin pixeli este legată de imposibilitatea determinării extremelor funcției pe caz general. Viteza redusă de trasare face ca pentru curbele uzuale (dreapta, cerc etc.) să se folosescă algoritmi particulari, care speculează simetria curbelor respective.

Exercițiu

Să se realizeze o aplicație ce împarte static fereastra formei în două coloane, fiecăreia corespunzându-i câte un control de tip **panel** şi cărora le asociază vizualizări distincte:

- panelului stâng i se asociază o vizualizare de **tip formular** ce permite introducerea a patru valori pozitive;
- vizualizarea din dreapta se foloseşte pentru **afişarea grafică** a celor patru valori. Selectarea tipului de grafic (*coloane*, *linii*, *pie*) se face folosind meniul aplicației. Modificarea culorii folosite pentru reprezentarea fiecărei valori se face prin click dreapta pe *TextBox*-ul corespunzător.

Rezolvare

Controlul de tip **panel** funcționează ca un container pentru celelalte controale.

1. Se crează un proiect *Visual C#*, cu tip aplicație *Windows Application.*
2. Se inserează un control de tip **panel (ToolBox / Windows Forms / Panel**), pe jumătatea stângă a ferestrei formular, proprietatea **Layout / Dock / Left** (controlul de tip **panel** este lipit de partea stângă a ferestrei aplicației), proprietatea **Layout / AutoScroll** valoarea **`True`** (barele de scrol apar automat dacă controalele sunt plasate în afara zonei client vizibile a formei),
3. Se adaugă un control **Splitter (ToolBoox / WindowsForms / Splitter),** lipit de partea dreaptă a controlului de tip **panel** (se modifică proprietatea **Layout / Dock** în **Left),**
4. Se adaugă un nou control de tip **Panel (Toolbox/Windows Forms / Panel**), care să umple întreaga fereastră formular rămasă, se modifică proprietatea **Layout / Dock** în **Fill,**
5. Se adaugă din **ToolBox** un control de tip **MainMeniu**: *Tip grafic,* având opțiunile *Coloane, Linii, Pie,* opțiunea *Coloane* - selectată implicit (proprietatea **Misc / Checked** valoarea **True**),
6. Se adaugă câte o pereche de controale **Label** și **TextBox** pentru cele patru valori, controalele de tip **TextBox** își modifică numele (proprietatea **Name**) în **tb_1, tb_2,...** ;
7. Se adaugă la proiect o nouă clasă - **`grafic`**, clasă care va modela datele: **ClassView /click dreapta / Add / Add Class/** `grafic`, având:

- **metodele:**
  - **`coloane, linii, pie`** – corespunzătoare celor trei opțiuni de trasare a graficului, prin bare verticale sau orizontale, respectiv grafic de structură în cerc;
  - **`puncte`** – pentru afișarea câte unui *x*, în fiecare punct de intersecție a două segmente, în graficul de tip *linie*,
  - **`Arata`** – pentru afișarea graficului corespunzător opțiunii selectate,
  - **`Maxim`** – pentru determinarea valorii maxime;
- **proprietățile** (de tip **`set`**) care permit stabilirea valorilor pentru: **`tip`** (tipul graficului de afișat) și **`culori`** (pentru păstrarea fiecărei valori a culorii).

Textul sursă al clasei **`grafic`**:


```
using System;
using System.Windows.Forms;
using System.Drawing;

namespace splitOK
{
 public class grafic
  {
   protected Pen pen;
   protected SolidBrush  brush;
   protected double [] v;
   protected double max;
   protected int nrval, tipgr, ddp, lp;
   protected Rectangle rect, rects;
   protected Point[] pct;
   protected Color []cul;

   public grafic()      {      }

   public grafic(Double[] val)
    {

      nrval = val.GetLength(0);
          // nr. valori de reprezentat
      v= new Double[nrval];
      v=val;
          // Pen pentru trasarea curbelor
      pen = new Pen(Color.Black);
          // Brush pentru colorarea suprafețelor
      brush= new SolidBrush(Color.White);
    }

   public int tip
     {  set  {  tipgr = value; }    }

   public Color [] culori
     {  set  { cul = value;    }    }


   public void Arata(PaintEventArgs e)
    {
      /*  PaintEventArgs  conține  clasa  Graphics  folosită
      pentru trasarea obiectelor pe dispozitivul de afișare
      precum  și  coordonatele  dreptunghiului  în  care  se  va
      desena  (stocat  de  proprietatea  ClipRectangle).  Un
      obiect  de  tip  Graphics  este  asociat  unui  device
      context. */
      rect=e.ClipRectangle;

      rect.X+=10; rect.Y+=10;
```

```
    rect.Height-=20; rect.Width-=20;
          // dreptunghiul care va încadra graficul
    e.Graphics.DrawRectangle
          (pen,rect.Left, rect.Top,rect.Width,rect.Height);
    switch(tipgr)
     {
           case 1: coloane(e); break;
           case 2: linii(e);   break;
           case 3: pie(e);     break;

     }
  }

public void coloane(PaintEventArgs e)
 {
  lp=(rect.Right-rect.Left)/3/(nrval+1); //lăţime dreptunghi
  ddp=(rect.Right-lp*nrval)/(nrval+1);
                          // distanţa dintre dreptunghiuri
  Maxim();
  for (int i=0; i<nrval;i++)
   {
    // se defineşte un dreptunghi corespunzător valorii i
    rects= new Rectangle(
        rect.Left+ddp*(i+1)+lp*i,
        (int)(rect.Bottom-v[i]*(rect.Bottom-rect.Top)/max),
        lp,(int)(v[i]*(rect.Bottom-rect.Top)/max)       );
     // Culoare extrasa din vectorul de tip Color
     brush.Color = cul[i];
     // desenare bara i
     e.Graphics.DrawRectangle(pen,rects);

     // se colorează dreptunghiul i
     e.Graphics.FillRectangle(brush,rects);
    }
 }

public void linii(PaintEventArgs e)
  {
    ddp=rect.Right/(nrval+1); // distanţa dintre 2 puncte
    Maxim();
    pct= new Point[nrval];
    for (int i=0; i<nrval;i++)
     {
        pct[i].X=rect.Left+ddp*(i+1);
        pct[i].Y=(int)
           (rect.Bottom-v[i]* (rect.Bottom-rect.Top)/max);
        // se trasează, câte un x
        puncte(pct[i],cul[i],e);
     }
        // se trasează segmentele de dreaptă intre
```



```
        // punctele cu coordonatele în vectorul de
        // structuri de tip Point
        e.Graphics.DrawLines(pen,pct);
  }

public void pie( PaintEventArgs e )
 {
     float start=0F;
     /* unghiul, măsurat în grade (în sensul acelor de ceas)
     de la axa OX până la prima latură a feliei de pie */
     float stop;
     /*unghiul, masurat in grade (in sensul acelor de ceas)
     de la axa OX până la ultima latură a feliei de pie  */
     double sum=0;
     for (int i=0; i<nrval;i++)
      {
          sum += v[i];
      }

     for (int i=0; i<nrval; i++)
      {
          // unghiul e proporţional cu valoarea
       stop=(float)(360F*v[i]/sum);
          // se trasează o felie din pie
       e.Graphics.DrawPie(pen,rect.X,rect.Y,rect.Width,
                         rect.Height,start, stop );
       brush.Color=cul[i]; // se colorează felia
       e.Graphics.FillPie(brush,rect.X,rect.Y,rect.Width,
                   rect.Height, start, stop      );
       start+=stop;
       // unghiul de start al următoarei felii este egal cu
       // cel de stop al celei curente
        }
  }

public void Maxim ()
  {

     max=v[0];
     for(int i=1;i<nrval;i++)
       if(max<v[i]) max=v[i];
  }

/* în punctul de intersecţie a doua segmente de dreaptă se
trasează un X, cu culoarea corespunzătoare functiei */

public void puncte(Point p, Color c, PaintEventArgs e)
 {
     pen.Color=c;
```

```
      e.Graphics.DrawLine(pen,p.X-3,p.Y-3, p.X+3, p.Y+3);
      e.Graphics.DrawLine(pen,p.X+3, p.Y-3, p.X-3,p.Y+3);
      pen.Color=Color.Black;
   }
 }
```

În clasa **Form1**:

```
public class Form1 : System.Windows.Forms.Form
{
//...
 grafic mygr;
 int nrv, men;
 double [] date;
 private System.Windows.Forms.TextBox [] vtb;
 // vector de referinte TextBox-uri
 Color [] csel;

 public Form1()
  {
   InitializeComponent();
   nrv=4; // numărul valorilor
   vtb= new System.Windows.Forms.TextBox[nrv];
   vtb[0]=tb_1; vtb[1]=tb_2; vtb[2]=tb_3; vtb[3]=tb_4;
   csel = new Color[nrv]; // vector de culori
   for (int i=0;i<nrv;i++)
      {
       vtb[i].Text="0";
       csel[i]=vtb[i].ForeColor;
      }
    men=1;
  }
.........
```

8. Se tratează evenimentul **Paint** al panelului 2: **Panel2/Events/Appearance / Paint**. Acest eveniment este declanşat ori de câte ori trebuie retrasat **panel2.**

```
private void panel2_Paint(object sender, PaintEventArgs e)
 {
  date =new  Double [nrv];
  for(int i=0;i<nrv;i++)
   {
      // preluare date din textBox-uri
      date[i]=Convert.ToDouble(vtb[i].Text);
   }

  // dacă s-a introdus cel puţin o valoare diferită de zero
  if((date[0]!=0)|(date[1]!=0)|(date[2]!=0)|(date[3]!=0))
```



```
   {
     mygr=new grafic(date); // instantiaza obiect grafic
     mygr.tip = men; // tip grafic preluat din meniu
     mygr.culori=csel; // preluare culoare selectata
     mygr.Arata(e); // se trasează graficul
   }
 }
```

9. Se tratează evenimentul **TextChanged**, pentru fiecare din cele patru controale de tip **TextBox**: proprietăţile controlului **TextBox / PropertyChanged / TextChanged**

```
private void tb_1_TextChanged(object sender, EventArgs e)
 {
     panel2.Invalidate(true); //generare mesaj Paint
 }
```

10. Se tratează evenimentul de redimensionare **panel2,** prin retrasare grafic: **Property Changed / SizeChanged**:

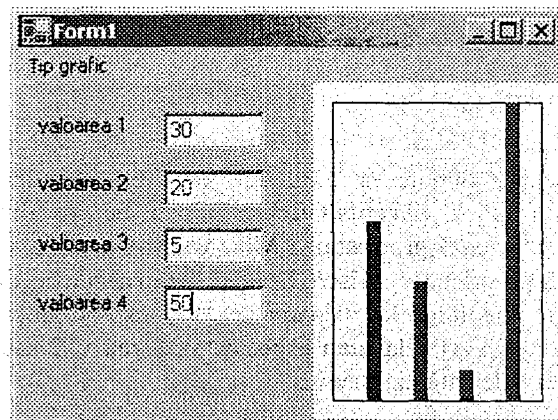
```
private void panel2_SizeChanged(object sender, EventArgs e)
 {
      Invalidate(true);
 }
```

11. Pentru fiecare opţiune a meniului ferestrei formular, se tratează evenimentul click **(Events/Click)**. Pentru fiecare dintre opţiunile meniului se setează proprietatea **Misc/RadioCheck** la valoarea **true**. Se setează proprietatea **Misc/Checked,** pentru fiecare dintre celelalte opţiuni ale meniului la valoarea **false**, iar opţiunea selectată ia valoarea **true.** Variabila **men** primeşte valoarea 1, 2 sau 3, în funcţie de opţiunea selectată din meniu.

```
private void menuItem2_Click(object sender, EventArgs e)
 {
     MenuItem item = (MenuItem)sender;
                          //itemul selectat al meniului
     Menu parent = item.Parent;
                          // părintele item-ului curent
     if ( item != null )
      {
           foreach ( MenuItem mi in parent.MenuItems )
                        mi.Checked = false;
           item.Checked = true;
      }
     men=item.Index+1;
     panel2.Invalidate(true);
 }
```

12. Din **ToolBoox/WindowsForms** se inserează un control de tip **ColorDialog**; se redenumeşte cu **cd**;
13. Pentru fiecare control **TextBox** se tratează evenimentul de click dreapta mouse, astfel: pe controlul **tb_1 /Events /Mouse/MouseDown,** se introduce textul sursă:

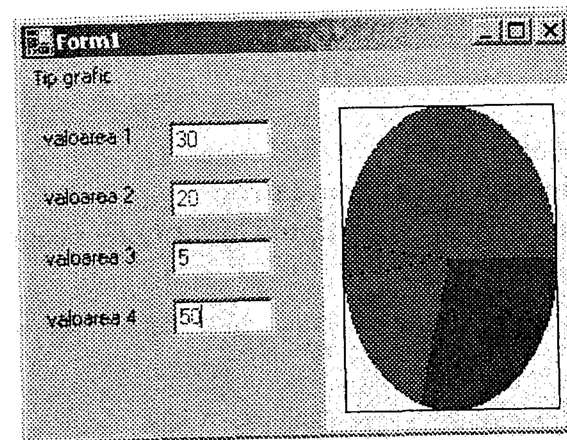
```
private void tb_1_MouseDown(object sender, MouseEventArgs e)
 {
  if(e.Button==MouseButtons.Right)
            // butonul dreapta mouse apăsat
   {
    cd.ShowDialog(); //  afişare ColorDialog
    csel[0]=cd.Color; // se preaia culoarea selectată
    tb_1.ForeColor=csel[0]; // colorează TextBox
    panel2.Invalidate(true);
   }
 }
```



`MouseEventArgs` specifică butonul mouse-ului care a fost apăsat, de câte ori a fost apăsat, coordonatele mouse-ului la click şi cu cât s-a mişcat rotiţa mouse-ului.

Proprietatea `Button` a variabilei **e** (de tip `MouseEventArgs`) indică butonul apăsat.

Enumerarea `MouseButtons` conţine constantele care definesc butonul care a fost apăsat.



Exerciţiu.

Să se construiască un ComboBox cu `DrawMode` de tip `OwnerDrawVariable` care să afişeze fonturile din sistem (`FontFamily.Families`) scrise cu propriu fontul. In acest sens, preluăm din argumentele evenimentului `DrawItem` **e.Graphics** şi scriem itemul cu `DrawString`.

Exerciţiu.

Să se construiască un ComboBox pentru alegerea unei culori, în care numele fiecărei culori este scris în culoarea respectivă.

LUCRU CU CLIPBOARD-UL. OPERAȚIA DE DRAG AND DROP

1. **Clipboard-ul**
2. **Operația de drag and drop**
3. **Drag and drop pe tipuri definite de programator**

1. Clipboard-ul

Transferul prin clipboard presupune ca obiectele să știe să se transforme în mai multe formate pentru a se adapta la capacitățile de înțelegere ale aplicației ce-l va prelua din clipboard.

Pe de altă parte, și aplicația care face *Paste*, când este scrisă de programator, poate fi și ea instruită să recunoască diferite formate de obiecte. Obiectele, de orice tip ar fi, se depun în clipboard printr-un apel **Clipboard.SetDataObject()**, ca de exemplu:

```
Clipboard.SetDataObject
      ("Text pus in Clipboard prin program");
```

sau:

```
Bitmap img = new Bitmap("sunset.jpg");
Clipboard.SetDataObject(img);
```

Transferul de date în Windows Forms este permis pentru clasele care implementează interfața **IDataObject;** acest lucru face ca toate obiectele să aibă o parte constituită din metodele de interfață standard moștenite, iar pe de altă parte programatorul să poată gestiona orice obiect prin referință la interfața de bază moștenită.

Aplicația care folosește obiectele puse în clipboard cere mai întâi o referință la "subobiectul" **IDataObject,** invocând pentru aceasta metoda **Clipboard.GetDataObject():**

```
IDataObject o = Clipboard.GetDataObject();
```

Dispunând de referința la această interfață, aplicația se poate interesa acum dacă cea ce există la un moment dat în clipboard este de un anume tip de obiect sau se poate converti într-un anume tip; pentru aceasta, invocă metoda **GetDataPresent( ),** căreia îi furnizează ca parametru tipul căutat :


```
bool vb = o.GetDataPresent(typeof(Bitmap));
```

iar metoda răspunde afirmativ sau nu. În acest fel, aplicația consumatoare își extrage din clipboard doar obiectele pe care le recunoaște ca tip sau acestea sunt capabile să se transforme într-un format acceptat de aplicația consumatoare.

Nu trebuie uitat că sub .NET orice obiect derivat din object știe să se transforme în `string` (metoda `ToString()`) și că în ultimă instanță am găsi ceva de pus ca răspuns la comanda de **Paste**, dacă acest lucru ne satisface. Pe de altă parte, nu știm cine a încărcat clipboard-ul și deci pot exista acolo și obiecte ce nu există în .NET.

Dacă interogând interfața **IDataObject**, printr-un apel **GetDataPresent(typeof(tip_dorit))**, implementată într-o formă specifică și de obiectul aflat în Clipboard, ni se răspunde afirmativ cu privire la existența acelui tip, se poate lansa o cerere de extragere sau conversie a obiectului din clipboard, folosind metoda **GetData( )** .

```
if(o.GetDataPresent(typeof(Bitmap)))
 {
      Bitmap img = (Bitmap)o.GetData(typeof(Bitmap));
      g.DrawImage(img,panel1.ClientRectangle);
 }
```

Exercițiu

Să se scrie un exemplu în care prin două butoane inscripționate **"Pune text in Clipboard"** și **"Pune bmp in Clipboard"** se încarcă prin program clipboard-ul cu un text sau cu o imagine.

Prin intermediul altui buton, inscripționat `"Scoate din Clipboard"`, conținutul clipboard-ului este descărcat într-un panel aflat pe forma principală a aplicației.

Rezolvare

Se aduc din ToolBox cele trei butoane și panelul, punându-li-se proprietăți adecvate. Programul ar putea arăta astfel :

```
using System;
using System.Drawing;
using System.Windows.Forms;

class Test : Form
{
 private System.Windows.Forms.Button b1;
```

```
private System.Windows.Forms.Button b2;
private System.Windows.Forms.Button b3;
private System.Windows.Forms.Panel panel1;
Test()
 {
     InitializeComponent();
 }

public void PuneText(Object s, EventArgs a)
{
 Clipboard.SetDataObject
           ("Text pus in Clipboard prin program");
 }

private void PuneBmp(object sender, System.EventArgs e)
 {
     Bitmap img=new Bitmap("sunset.jpg");
     Clipboard.SetDataObject(img);
 }


public void Scoate(Object s, EventArgs a)
 {
     Graphics g = Graphics.FromHwnd(panel1.Handle);
     IDataObject o = Clipboard.GetDataObject();

     if(o.GetDataPresent(typeof(string)))
      {
       g.DrawString(o.GetData(typeof(string)).ToString(),
                 Font, new SolidBrush(Color.Black),10,10  );
      }
     else if(o.GetDataPresent(typeof(Bitmap)))
      {
            Bitmap img = (Bitmap)o.GetData(typeof(Bitmap));
            g.DrawImage(img,panel1.ClientRectangle);
      }
     else MessageBox.Show
            ( "Format obiect din Clipboard nerecunoscut");
 }
private void InitializeComponent()
 {
  this.b1 = new System.Windows.Forms.Button();
  this.b2 = new System.Windows.Forms.Button();
  this.panel1 = new System.Windows.Forms.Panel();
  this.b3 = new System.Windows.Forms.Button();
  this.SuspendLayout();
            //
            // b1
            //
```



```
this.b1.Location = new System.Drawing.Point(8, 72);
this.b1.Name = "b1";
this.b1.Size = new System.Drawing.Size(128, 23);
this.b1.TabIndex = 0;
this.b1.Text = "Pune text in Clipboard";
this.b1.Click += new System.EventHandler(this.PuneText);
          //
          // b2
          //
this.b2.Location = new System.Drawing.Point(424, 128);
this.b2.Name = "b2";
this.b2.Size = new System.Drawing.Size(120, 23);
this.b2.TabIndex = 1;
this.b2.Text = "Scoate din Clipboard";
this.b2.Click += new System.EventHandler(this.Scoate);
          //
          // panel1
          //
this.panel1.BackColor = System.Drawing.Color.FromArgb(
                ((System.Byte)(255)),
                ((System.Byte)(255)),
                ((System.Byte)(192)));
this.panel1.Location = new System.Drawing.Point(144, 0);
this.panel1.Name = "panel1";
this.panel1.Size = new System.Drawing.Size(272, 272);
this.panel1.TabIndex = 2;
          //
          // b3
          //
this.b3.Location = new System.Drawing.Point(8, 184);
this.b3.Name = "b3";
this.b3.Size = new System.Drawing.Size(128, 23);
this.b3.TabIndex = 3;
this.b3.Text = "Pune bmp in Clipboard";
this.b3.Click += new System.EventHandler(this.PuneBmp);
          //
          // Test
          //
this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
this.ClientSize = new System.Drawing.Size(560, 273);
this.Controls.Add(this.b3);
this.Controls.Add(this.panel1);
this.Controls.Add(this.b2);
this.Controls.Add(this.b1);
this.Name = "Test";
this.ResumeLayout(false);
}

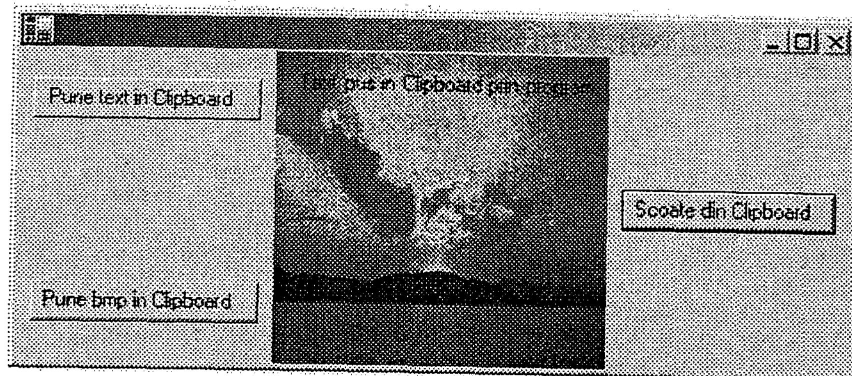
public static void Main()
 {
```

```
        Application.Run(new Test());
    }
}
```

Existența celor două butoane pentru a încărca prin program clipboard-ul cu ceva, nu exclude posibilitatea preluării în clipboard a unor obiecte prin intermediul altei aplicații ; spre exemplu, **din Word se pot prelua texte sau imagini** (bmp sau ico) cu **Copy**, urmând ca în aplicația noastră să se apese **doar pe butonul de scos din clipboard**. În cazul în care clipboard-ul conține obiecte de tip necunoscut și care nici nu știu să se convertească într-un tip recunoscut de aplicația noastră, se va afișa un mesaj în acest sens (spre exemplu, preluați cu *Copy* un control din Designer și încercați să faceți *Paste* în aplicația de mai sus).

Invers, imaginea sau textul puse în clipboard prin apăsarea butoanelor aplicației pot fi inserate cu **Paste** într-un document Word sau într-o altă aplicație care înțelege aceste formate.

Chiar dacă obiectul din clipboard este de un tip acceptat de aplicația consumatoare, la apelul `GetData` este necesară și aplicarea unui cast, valorii returnate de funcția de extragere.



2. Operația de drag and drop

Inițierea operației de **drag & drop** se poate face de pe orice eveniment, însă uzual ea se sincronizează cu evenimentul **`MouseDown`**, deoarece acesta furnizează informații detaliate despre poziția mouse-ului. Inițierea propriu-zisă se face printr-un apel **`DoDragDrop`**, recunoscută de multe controale. În acest moment se stabilesc datele ce fac obiectul dragării, precum și drepturile acordate de sursă asupra datelor: copiere sau mutare.

Următorul eveniment ce intervine în operația de drag & drop este **`DragEnter`**, declanșat la intrarea în regiunea unui control care permite droping (proprietatea **`AllowDrop`** este **`true`**). Cu acest prilej se consultă drepturile permise de această posibilă destinație și acestea sunt confruntate cu cele acordate de sursă, determinându-se efectul final pentru drop.

Cel de-al treilea eveniment, **`DragDrop`** se generează în momentul relaxării mouse-ului în timpul derulăriiunei operații de dragare. Cu acest prilej, se furnizează codul care materializează efectul operației.

Obiectul transmis ca parametru în funcțiile de tratare a evenimentelor implicate (blocul de argumente) ține și datele implicate în operația de drag & drop

Unele controale recunosc chiar **evenimente specifice** legate de operația drag & drop; de exemplu, `ListView` și `TreeView` au evenimentul **`ItemDrag`**.

Exercițiu. Să se încarce un TextBox preluând prin drag and drop, textul înscris pe un buton.

1. Se crează o aplicație formular;
2. se aduc din ToolBox un buton și un textbox ;
3. se marchează textbox-ul ca acceptând drop (**Properties / `AllowDrop`** pe **`true`**);
4. se inițiază operația de drag & drop pe evenimentul **MouseDown** de buton, moment în care printr-un apel **`DoDragDrop`** se fixează sursa de date ce face obiectul dragării (textul înscris pe buton) și efectele permise pentru destinație (`Copy` sau `Move`).
   Observație:
   `DoDragDrop` este o metodă de-a controlului sursă (butonul, în cazul nostru) !

```
private void button1_MouseDown(object sender,
                System.Windows.Forms.MouseEventArgs e)
{
  button1.DoDragDrop(button1.Text,
      DragDropEffects.Copy | DragDropEffects.Move  );
}
```

5. la intrarea mouse-lui pe suprafața textbox-ului (evenimentul **DragEnter**) se testează compatibilitatea formatului datelor dragate cu formatul destinației posibile. În caz de neconcordanță se pot face conversii pe acest eveniment sau pur și simplu se forțează efectul **`None`**, în loc de **`Copy`** sau **`Move`**.

```
private void textBox1_DragEnter(object sender,
                System.Windows.Forms.DragEventArgs e)
{
      if (e.Data.GetDataPresent(DataFormats.Text))
                e.Effect = DragDropEffects.Copy;
      else
                e.Effect = DragDropEffects.None;
}
```

6. textbox-ul acceptă ***drop***, schimbându-şi astfel textul prin preluarea celui indicat prin argumentul funcţiei de tratare a evenimentului DragDrop:

```
private void textBox1_DragDrop(object sender,
                System.Windows.Forms.DragEventArgs e)
{
  textBox1.Text =
          e.Data.GetData(DataFormats.Text).ToString();
}
```

Se pot transfera date încapsulate foarte variate, spre exemplu chiar cod sursă necesar regăsirii datelor într-un DataSet.

Pe **DragEnter** sau **DragOver** se stabileşte şi efectul (acţiunea) care se va aplica la drop în acest caz, unul dintre cele fixate iniţial de către sursă, la apelul **DoDragDrop**. Efectele permise de sursă pot fi consultate acum prin proprietatea **e.AllowedEffect** numai că sunt stocate pe biţi, corespunzători; spre exemplu expresia **e.AllowedEffect & DragDropEffects.Copy** izolează bitul aferent efectului de Copy, bit ce poate fi testat comparându-l cu constantele posibile (în cazul nostru DragDropEffects.Copy.

Efectul de executat se poate stabili dinamic şi în funcţie de starea combinată a altor taste (CTRL, SHIFT, ALT) sau a butoanelor mouse-ului. Proprietatea **e.KeyState** ţine câte un bit de stare pentru fiecare tastă sau buton de mouse, putând indica prezenţa cumulativă a mai multor taste apăsate; valorile individuale asociate sunt indicate mai jos.

1	buton stânga mouse
2	buton dreapta mouse.
4	tasta SHIFT
8	tasta CTRL
16	buton mijloc mouse
32	tasta ALT

Dacă în funcţia textBox1_**DragEnter** de mai sus schimbăm codul astfel:

```
private void textBox1_DragEnter(object sender,
       System.Windows.Forms.DragEventArgs e)
 {
   e.Effect = DragDropEffects.None;
      // implicit, nimic fara permisiune

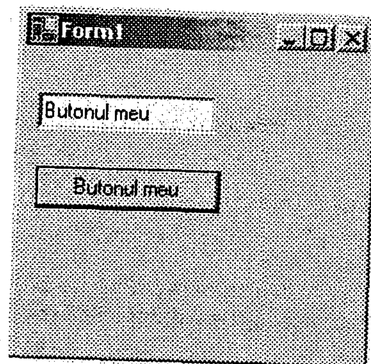
    if ((e.KeyState & 8) == 8 &&  // bitul CTRL este setat
       (e.AllowedEffect & DragDropEffects.Copy) == // izolare
                                  DragDropEffects.Copy)
                      // are si permisiune (bitul) Copy
          e.Effect = DragDropEffects.Copy; // face Copy
    else
       if ((e.AllowedEffect & DragDropEffects.Move) ==
                                      DragDropEffects.Move)
            // altfel Move, daca bitul Move permis
          e.Effect = DragDropEffects.Move;
 }
```

atunci prin apăsare CTRL se indică drept efect **copierea** textului în TextBox (textul rămâne şi pe buton). De remarcat că deoarece tratarea s-a făcut pe evenimentul **DragEnter** şi nu **DragOver,** tasta CTRL trebuie să fie apasată înainte de intrarea cu mouse-ul în regiunea butonului, altfel efectul nu mai poate fi schimbat, rămânând activ cel de *Move*.

În funcţia textBox1_DragDrop se stabileşte în clar în ce constă acţiunea asociată cu drop, având grijă ca în cazul efectului *Move* să anulăm şi textul de pe buton:

```
textBox1.Text = e.Data.GetData(DataFormats.Text).ToString();
if(e.Effect==DragDropEffects.Move)
      button1.Text = "";
```

Drag & drop între aplicaţii se execută după aceleaşi reguli; fără a modifica nimic în codul programului, la rulare redimensionăm ferestrele Visual Studio şi cea a aplicaţiei, încât să fie vizibile ambele simultan. Facem o dragare de text din editorul de text Visual Studio în textbox-ul aplicaţiei şi vedem că se execută în aceeaşi manieră ca şi cum am lucra în interiorul unei aplicaţii.



3. Drag and drop pe tipuri definite de programator

Ca şi transferul prin clipboard-ul, facilitatea de drag and drop se bazează în .NET tot pe interfaţa **IDataObject** şi realizează o formă simplă vizuală de transfer al obiectelor, între aplicaţii sau între componentele aceleeaşi aplicaţii. Ca urmare, fie se transferă obiecte uzuale de tipul `String`, `Image` etc., fie se definesc clase care implementează interfaţa **IDataObject**, prin care obiectele sunt învăţate cum să se transforme şi în alte tipuri, recunoscute şi acceptate de alte aplicaţii.

Exerciţiu

Să se definească o clasă **Pers** capabilă să suporte drag and drop, avînd ca destinaţii posibile aplicaţii sau componente care acceptă tipurile `Bitmap`, `String`, `ListViewItem` sau `PrintDocument`.

Rezolvare

Clasa trebuie să fie derivată din **IDataObject** şi va conţine câteva câmpuri membre printre care şi numele unui fişier ce conţine poza persoanei respective.

```
public class Pers:IDataObject
  {
  // Aici se introduç metodele interfeţei IDataObject

        public  int Marca;
        public string Nume;
        string fisImagine;

        public Pers(int m, string n, string fisImg)
              {Marca=m; Nume=n;fisImagine=fisImg;}

  }
```


Implementarea interfeţei **IDataObject** de către o clasă de utilizator trebuie să rezolve patru probleme:

1. să expună formatele suportate de clasă;
2. să încarce datele furnizate de clasă ;
3. să dea răspunsul la întrebarea dacă obiectul se poate transforma sau nu într-un format anume, cerut de o aplicaţie client;
4. să extragă datele în formatul adecvat fiecărei aplicaţii client.

Ca urmare vor trebuie adăugate clasei Pers metodele specifice de implementare a interfeţei **IDataObject**, care să rezolve toate aceste probleme.

1. Expunerea formatelor suportate de clasă

La prima problemă, răspunsul îl dă scrierea supraîncărcărilor funcţiei **`GetFormats()`**. Acestea returnează un vector de formate în care un **DataObject** este disponibil, deci numele claselor .NET în care datele conţinute de obiect pot fi convertite. În acest sens, clasa `Pers` va conţine două versiuni ale funcţiei **`GetFormats( )`**.

Prima versiune returnează doar tipurile native în care ea se poate converti; aceste tipuri pot fi clase .NET, alte clase introduse de utilizator sau interfeţe din care e derivată clasa şi reprezintă formatele în care data este stocată :

```
public string[] GetFormats()
  {
        return new string[]
              {  "Pers", "Bitmap","listViewItem"  };
  }
```

Cea de-a doua versiune primeşte un parametru de intrare de tip `bool`, indicând dacă utilizatorul doreşte sau nu şi tipuri alternative, altele decât cele native, în care obiectul dragat s-ar putea converti dacă tipul solicitat nu este disponibil; **`autoConvert`** pe **`false`** foloseşte când dorim să nu permitem o anumită conversie de adaptare (spre exemplu, tipul `UnicodeText` stocat s-ar putea la rândul lui converti la nevoie în `Text` sau în `String`), ci să folosim doar formatele native.

```
public string[] GetFormats(bool autoConvert)
 {
  if (autoConvert) // toate formatele
```

```
    return new string[]
      { "Pers", "Bitmap","ListViewItem", DataFormats.Text };
                        // total tipuri: native + generale
  else
    return GetFormats();// doar cele native
}
```

2. Încărcarea datelor furnizate de clasă

Se face prin versiunile supraîncărcate ale funcţiei **SetData**; prima este varianta de bază şi încarcă pur şi simplu o referinţă la obiectul dragat.

```
object obiectul; // referinta obiectului de transferat

public void SetData (object o)
      {
            if (o is Pers)    {obiectul = (Pers) o;}
            // obiectul propriu-zis
      }
```

Următoarele două versiuni cer stocarea într-un anumit format, acesta fiind indicat ca prim parametru de apel, fie prin denumire, fie ca tip recunoscut sub .NET :

```
//.SetData pentru tipurile native
public void SetData(string fmt, object o)
 {
  switch(fmt)
  {
      case "Pers":              SetData(o);break;
      case "ListViewItem":      SetData(o);break;
      case "Bitmap":            SetData(o);break;
      case "Text":
            obiectul = new Pers(0,"xxx","");break;
      default: MessageBox.Show
            ("Nu se poate converti in "+fmt);break;
  }
 }

      // sa mearga si sub forma: in ce tip, care obiect
public void SetData(Type t, object o)
            {     SetData(t.Name, o);     }
```

După cum se vede, ultimele două versiuni fac apel, direct sau indirect, tot la forma de bază.

O altă versiune permite setarea obiectului într-un format depinzând de un al treilea parametru, cel de autoconvert, cu semnificaţia explicată mai sus, la funcţia *GetFormats*( ).

```
public void SetData(String fmt, bool convert, object o)
                  // supraincarcare cu autoconvert
 {
      if (fmt == DataFormats.Text&& convert == false )
                                    // doar formatele stocate
            {
             MessageBox.Show
              ("Refuz prin parametru sa convertesc in "+fmt);
            }
      else
      //chiar si in formate in care stie sa se autotransforme
            {
                  SetData(fmt, o);
            }
 }
```

Dacă se doreşte stocarea datelor în mai multe formate se invocă de mai multe ori SetData cu aceeaşi instanţă de IDataObject ;

3. Răspunsul la întrebarea dacă obiectul se poate sau nu transforma într-un format anume, dorit de o aplicaţie client

Este furnizat de apelul uneia din versiunile funcţiei **GetDataPresent()**:

```
public bool GetDataPresent(string fmt)
      {
            if (fmt == "Pers" || fmt == "Bitmap"
                        || fmt == "ListViewItem"
                        || fmt == DataFormats.Text)
                  return true;
            else
                  return false;
      }

public bool GetDataPresent(Type t)
      { //.acelasi lucru, dar cu tip dat ca   parametrul
                  return(GetDataPresent(t.Name));
      }     // redirecteaza la cea de mai sus

            // acelasi, dar si cu optiune de autoconvert
public bool GetDataPresent(String fmt, bool convert)
  {
      if (fmt == DataFormats.Text && convert == false)
```

```
        // variantele stocate, fara conversie in alte tipuri
            return false;
        else
        // cu posibile conversii in alte tipuri
            return GetDataPresent(fmt);
}
```

Se observă că interogarea se poate face dând denumirea formatului ca string, sau tipul obiectului, făcând sau nu distincție între formatele de stocare și cele obținute prin conversii suplimentare.

4. Extragerea datelor în formatul adecvat fiecărei aplicații client

Extragerea datelor se face printr-un apel GetData(); să ne reamintim că instanțe ale clasei **DataObject** care mediază transferul prin drag and drop, sunt stocate în diverse formate la o referință gestionată printr-o referință de object generic, denumită de noi `obiectul`. Conversia se poate face așadar la sursă (înainte de încărcarea referinței prin SetData) sau la destinație, la momentul extragerii obiectului prin GetData( ). Versiunea propusă aici pentru GetData() face conversii la destinație pentru toate formatele.

```
public object GetData(string fmt)
{
 switch (fmt)
     {
        case "Bitmap" :
            return new Bitmap( ((Pers)obiectul).fisImagine );

        case "ListViewItem" :// conversie la destinatie
            ListViewItem itm =
               new ListViewItem(""+((Pers)obiectul).Marca,1);
            itm.SubItems.Add(((Pers)obiectul).Nume);
            return itm;

        case "Pers" : // nu necesita conversie
            return obiectul;

        case "Text" : // conversie la destinatie
            return    "******************\r\n"+
                      ((Pers)obiectul).Marca+ "  "
                      +((Pers)obiectul).Nume+
                      "\r\n******************";
     default :
        MessageBox.Show("Ar putea fi, dar nu e inca "+fmt);
            return null;
     }
}
```


Elemente pregătitoare în cadrul aplicației

În clasa `Form1` se declară colecțiile de persoane, de `PictureBox` și o listă de imagini.

```
public ArrayList persColect;
public ArrayList pictColect;
public Pers [] vp;
public PictureBox [] vi;
ImageList li;
```

S-a preferat încărcarea colecțiilor de persoane și de imagini prin preluarea elementelor din vectorii `vp`, respectiv `vi`. S-au ales colecții în locul vectorilor pentru a putea elimina elemente pe măsura dragării unor iconuri în *Recycle Bin*.

În constructorul clasei `Form1` au fost încărcate toate aceste colecții:

```
public Form1()
 {
      InitializeComponent();
      vp = new Pers[]
      {
            new Pers(0,"Zero ","fis0.bmp"),
            new Pers(1,"Unu ","fis1.bmp"),
            new Pers(2,"Doi ","fis2.bmp"),
            new Pers(3,"Trei","fis3.bmp"),
            new Pers(4,"Patru","fis4.bmp"),
            new Pers(5,"Cinci","fis5.bmp")
      };
      persColect = new ArrayList(vp);

      vi = new PictureBox[]
      {
            pictureBox3,pictureBox4,pictureBox5,
            pictureBox6,pictureBox7,pictureBox8
      };
      pictColect=new ArrayList(vi);

      li=new ImageList();
            li.Images.Add( new Icon("pers0.ico") );
            li.Images.Add( new Icon("pers1.ico") );
            li.Images.Add( new Icon("pers2.ico") );
            li.Images.Add( new Icon("pers3.ico") );
            li.Images.Add( new Icon("pers4.ico") );
            li.Images.Add( new Icon("pers5.ico") );

 }
```

Din punct de vedere vizual, au fost construite pe formă:

- un panel central ce conţine iconurile asupra cărora se execută operaţia de drag and drop;
- un control ListView, în dreapta, ce va prelua informaţiile prin dragarea iconurilor; drept cap de tabel, lista va afişa marca şi numele şi prenumele fiecărei persoane;
- un panel, în stânga, ce va permite vizualizarea bitmap-urilor (pozelor) persoanelor;
- două paneluri, în partea de jos, ce conţin imaginile unei imprimante şi a unui Recycle Bin şi permit implementarea prin acţiunea de drop, a previzualizării unui document cu cartea de vizită a unei persoane, respectiv, eliminarea unei persoane din colecţie.

Directorul ce va conţine fişierul executabil al aplicaţiei a fost populat cu câteva fişiere denumite **persnn.ico**, respectiv **fisnn.bmp**, ce conţin icon-uri sugerând persoane şi imagini bmp ale unor persoane, câte un fişier din fiecare categorie pentru fiecare persoană din colecţie.

Încărcarea iconurilor s-a făcut printr-o funcţie specializată:

```
public void IncarcaIconuri()
{
	int poz=0;
	foreach(Image i in li.Images)
	{
		PictureBox pb =(PictureBox)pictColect[poz];
		pb.Image=li.Images[poz];
		poz++;
	}
}
```

apelată atât la repictarea formei, cât şi la modificări aduse colecţiei de iconuri; la repictarea formei se adaugă peste paneluri şi celelalte imagini, preluate din fişiere aflate tot în directorul curent al aplicaţiei:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
	Graphics g = panel3.CreateGraphics();
	Bitmap cosul= new Bitmap("Cosul.bmp");
	g.DrawImage(cosul,panel3.ClientRectangle);

	g = panel4.CreateGraphics();
	Bitmap imprimanta= new Bitmap("imprimanta.bmp");
	g.DrawImage(imprimanta,panel4.ClientRectangle);
```



```
	IncarcaIconuri();
}
```

Iniţierea operaţiei de drag and drop

Se poate face pe evenimentul `MouseDown` asociat panelului cu iconuri. Cu acest prilej se stochează datele în diverse formate, folosind o instanţă fictivă a clasei `Pers`. Deşi nu se fac aici conversii, stocându-se de fapt de fiecare dată tot un obiect de tip `Pers`, apelurile repetate ale funcţiei au menirea să înregistreze existenţa diverselor tipuri de stocare.

```
private void MouseDownIconuri(object s, MouseEventArgs e)
 {    // initiere d&d

  Pers persSursa=new Pers(0,"Fictiv","");
  int poz= pictColect.IndexOf((PictureBox)s );
  if(poz>-1 && poz <persColect.Count)
   {
      // cate un SetData pt fiecare tip, pe acelasi obiect!!
      persSursa.SetData(persColect[poz]);
      persSursa.SetData("Text",persColect[poz]);

      persSursa.SetData("ListViewItem",persColect[poz]);

      persSursa.SetData("Bitmap",persColect[poz]);
   }
  ((Control) s).DoDragDrop
    ( persSursa, DragDropEffects.Copy|DragDropEffects.Move );
  // controlul permite efect de Copy sau Move(autoreflexie)
  }
```

Tratarea evenimentului de drag and drop la nivelul destinaţiei

Obiectele destinaţie trebuie să aibă proprietatea `AllowDrop` pe `true`. Pentru fiecare din posibilele destinaţii care acceptă drag and drop, se scrie câte o pereche de funcţii:

- **xxx_DragEnter**, în care se testează dacă obiectul dragat acceptă unul din formatele recunoscute şi de obiectul deasupra căruia se află, moment în care se decide asupra efectului de urmat, `None` în cazul incompatibilităţii de formate.
- **xxx_DragDrop**, în care se materializează efectul dorit pentru drop.

Pentru controlul **ListView**, cele două funcţii sunt

```
private void listView1_DragEnter
      (object sender, System.Windows.Forms.DragEventArgs e)
  {
         if (e.Data.GetDataPresent("ListViewItem")  )
               e.Effect = DragDropEffects.Copy;
         else
               e.Effect = DragDropEffects.None;
  }

private void listView1_DragDrop
      (object sender, System.Windows.Forms.DragEventArgs e)
 {
      listView1.Items.Add(
            (ListViewItem)e.Data.GetData("ListViewItem") );
 }
```

Pentru panelul ce conține un bitmap asociat fiecărei persoane, funcția de drop face extragerea obiectului în format **Bitmap** și vizualizează imaginea.

```
private void panel2_DragEnter
      (object sender, System.Windows.Forms.DragEventArgs e)
  {
         if (e.Data.GetDataPresent("Bitmap")  )
               e.Effect = DragDropEffects.Copy;
         else
               e.Effect = DragDropEffects.None;
  }

         // drop genereaza bitmap
private void panel2_DragDrop
      (object sender, System.Windows.Forms.DragEventArgs e)
 {
      Graphics g = panel2.CreateGraphics();
      Bitmap b=(Bitmap)e.Data.GetData("Bitmap");
      g.DrawImage(b,panel2.ClientRectangle);
 }
```

Pentru panelul ce semnifică trimiterea unui icon în **Recycle Bin**, funcția de drop extrage obiectul dragat, îi identifică poziția în colecția de persoane, după care luând ca reper această colecție operează eliminările poziției respective din colecția de persoane și din lista de iconuri, iar în colecția de PictureBox-uri, o face pe ultima invizibilă și redistribuie prin reîncărcarea pe primele locuri, imaginile rămase.

```
private void panel3_DragEnter
```



```
      (object sender, System.Windows.Forms.DragEventArgs e)
 {
      if (e.Data.GetDataPresent("Pers")  )
            e.Effect = DragDropEffects.Move;
      else
            e.Effect = DragDropEffects.None;
 }

// arunca la cos
private void panel3_DragDrop
      (object sender, System.Windows.Forms.DragEventArgs e)
 {
    Pers p = (Pers)e.Data.GetData("Pers");
    int poz =persColect.IndexOf(p);
          // persColect este reperul pt toate colectiile

    persColect.RemoveAt(poz); // scot persoana

    //ultimul box devine invizibil;
   ((PictureBox)pictColect[persColect.Count]).Visible=false;
   li.Images.RemoveAt(poz); // scot iconul din lista
   IncarcaIconuri(); // reincarca icon-urile
 }
```

Dragarea peste panelul ce afișează o **imprimantă** este interpretată ca o cerere de previzualizare a cărții de vizită a persoanei dragate. În acest sens, funcția de tratare va extrage obiectul sub formă de text, îl va stoca într-o variabilă alocată la nivelul formei, după care va instanția un `PrintDocument` și un `PrintPreviewDialog`, și va face legătura între ele declarând documentul ca proprietate a dialogului de previzualizare. Imprimarea propriu-zisă o face o funcție (**`prtDoc_PrintPage`**)de tratare a evenimentului *PrintPage*, declanșat automat odată cu activarea dialogului de previzualizare.

```
private void panel4_DragEnter(object sender,
            System.Windows.Forms.DragEventArgs e)
 {
       if (e.Data.GetDataPresent("Pers")  )
             e.Effect = DragDropEffects.Copy;
       else
             e.Effect = DragDropEffects.None;
 }

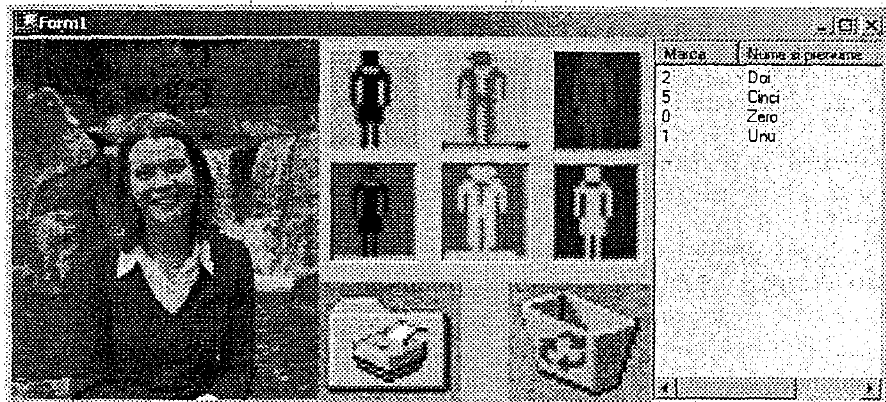
// drop genereaza preview
private void panel4_DragDrop(object sender,
            System.Windows.Forms.DragEventArgs e)
 {
```

```
    txt=(String)e.Data.GetData("Text");
    PrintPreviewDialog prevDial=new PrintPreviewDialog();
    System.Drawing.Printing.PrintDocument prtDoc;
    prtDoc = new System.Drawing.Printing.PrintDocument();
    prtDoc.PrintPage += new
        System.Drawing.Printing.PrintPageEventHandler
                                            (prtDoc_PrintPage);
    prevDial.Document=prtDoc;
    prevDial.ShowDialog();
    panel2.Refresh();
}

private void prtDoc_PrintPage(object sender,
             System.Drawing.Printing.PrintPageEventArgs e)
{
    e.Graphics.DrawString(txt,new Font("Arial",12),
             new SolidBrush(Color.Red),10,10 );
    // txt membru al formei, contine  textul de afişat
}
```

Se poate testa făcând **dragarea unui icon peste un document MS Word**, observându-se că documentul va afişa obiectul `Pers` în format text.

Invers, preluând prin dragare o imagine bitmap dintr-un document Word şi lăsând-o peste panelul ce afişează bitmap-uri, acesta va reda imaginea dragată, confirmându-ne generalitatea standardului de implementare a interfeţei **IDataObject**.



LEGAREA DATELOR DE INTERFAŢA UTILIZATOR – DATA BINDING

1. **Legarea simplă – simple data binding**
2. **Legarea complexă - complex data binding**

1. Legarea simplă – simple data binding

De multe ori avem nevoie ca datele afişate de unele controale să reflecte valori ale unor variabile din program; spre exemplu, ne-ar interesa ca un TextBox să afişeze valoarea unei variabile şi setând textul în control să modificăm şi valoarea variabilei. Invers, dintr-o prelucrare rezultă o valoare modificată pentru variabilă şi este firesc TextBox-ul să afişeze noua valoare. Apare deci normală nevoia declarării unor astfel de legături, pentru a se putea alinia ambii membri ai legăturii, la modificarea unuia dintre ei. Legarea datelor presupune în acelaşi timp şi conversie automată, adică o legătură între reprezentarea internă a datelor în variabile şi reprezentarea lor externă, în controale de vizualizare.

O legătură poate fi **simplă** dacă de proprietatea unui control se leagă un singur element dintr-o colecţie de date, sau **complexă**, când controlul este capabil să afişeze toată colecţia de date.

Exerciţiu

Intr-o aplicaţie declarăm ca membru în formă, un vector de string **`vectFac`** şi-l iniţializăm pe evenimentul Load al formei sau în constructorul formei, cu denumirea unor facultaţi. Adăugăm pe formă şi un control TextBox numit **`txtFac`**.

Controalele conţin o colecţie de tip **`ControlBindingsCollection`** numită **`DataBindings`**. Ea ţine la rândul ei legări (obiecte de tip **`Binding`**) ale diverselor proprietăţi ale controlului, cu valori ale unor variabile aparţinând aşanumitelor tipuri „conectabile”.

Tot pe evenimentul Load al formei completăm colecţia cu o legătură între proprietatea **Text** a `txtFac` şi valorile din vectorul `vectFac`, astfel încât funcţia de tratare a evenimentului să arate astfel:

```
private void Form1_Load(object sender, System.EventArgs e)
{
  string [] vectFac =
                  { "CSIE","REI","COM","FIN","MAN","AGR"};
  txtFac.DataBindings.Add("Text",vectFac,"");
}
```

Se observă că adăugarea s-a făcut cu o versiune supraîncărcată a metodei **Add()**, fără crearea explicită a unui obiect de tip Binding; explicit aceeaşi adăugare se putea face în doi paşi:

```
Binding b= new Binding("Text",vectFac,"");
txtFac.DataBindings.Add(b);
```

La rulare constatăm că textBox-ul afişează deja denumirea primei facultăţi; mai mult decât atât, dacă modificăm textul din control, automat se modifică şi denumirea conţinută în vector, lucru ce se poate verifica dacă pe un buton **Afiseaza** se afişează într-un MessajBox primul element din vector, înainte şi după apăsarea pe butonul **Schimba**.

Legătura este însă unidirecţională, adică modificând prin program denumirea primei facultăţi, controlul nu îşi modifică automat textul afişat.

Tipurile conectabile la controale sunt:

- toate tipurile derivate din **IList**, inclusiv vectorii şi colecţiile, de tipuri fundamentale sau de tipuri introduse de utilizator;
- tipurile care implementează interfeţele **IBindingList** sau **ITypedList**, dintre care cele mai reprezentative sunt: **DataSet, DataTable, DataView, DataViewManager**.

Parametrii funcţiei **DataBindings.Add()** arată care proprietate a controlului este legată, de care variabilă, iar dacă variabila este o instanţă a unui tip introdus de utilizator, cel de-al treilea parametru precizează care anume **proprietate** (căci pot fi mai multe) a acestui tip va fi conectată cu controlul.

Spre exemplu, vom construi o clasă **Pers** care deţine două proprietăţi de tip read / write, pentru nume şi respectiv pentru vârstă. Se observă că **pot fi legate doar proprietăţi**, nu şi câmpuri simple, chiar dacă le declarăm publice.

De remarcat de asemenea, că proprietatea Varsta este asociată câmpului int varsta, deci legătura asigură automat şi conversia către textul conţinut de controlul txtVarsta.

```
class Pers
    {
        string nume;
        int varsta;
        public Pers( string n, int v) { Nume=n; varsta=v; }
```



```
        public string Nume
        {
            get { return nume;}
            set { nume=value; }
        }

        public int Varsta
        {
            get { return varsta;}
            set { varsta=value;}
        }
    }
```

iar funcţia care declară legăturile arată acum astfel:

```
private void Form1_Load(object sender, System.EventArgs e)
    {
        Binding b= new Binding("Text",vectFac,"");
        txtFac.DataBindings.Add(b);

        txtPers.DataBindings.Add("Text",vectPers,"Nume");
        txtVarsta.DataBindings.Add("Text",vectPers,"Varsta");
    }
```

De remarcat că pentru varietate, de această dată legarea controlului txtFac s-a făcut mai detaliat, instanţiind mai întâi un obiect de tip Binding, care a fost adăugat apoi la colecţia DataBindings a controlului.

In această formă, toate textBox-urile afişează doar prima componentă din vector; pentru a putea naviga prin vector se folosesc obiecte specializate: un **BindingContext** care gestionează mai multe obiecte de tip **CurrencyManager**, câte unul pentru fiecare sursă de date; la rândul lui, un obiect **CurrencyManager** ţine poziţia curentă într-o sursă de date, poziţie pe care o comunică tuturor controalelor legate la această sursă; cum este şi firesc, poziţia poate fi incrementată sau decrementată. Vom pune două butoane de navigare, inscripţionate cu "**<**"şi "**>**", pentru defilarea "înapoi" şi "înainte" în vector.

Funcţiile de tratare a Click-ului pe aceste butoane conţin doar câte o instrucţiune şi realizează modificarea poziţiei curente:

```
BindingContext[vectPers].Position -=1;
```

şi respectiv,

```
BindingContext[vectPers].Position +=1;
```

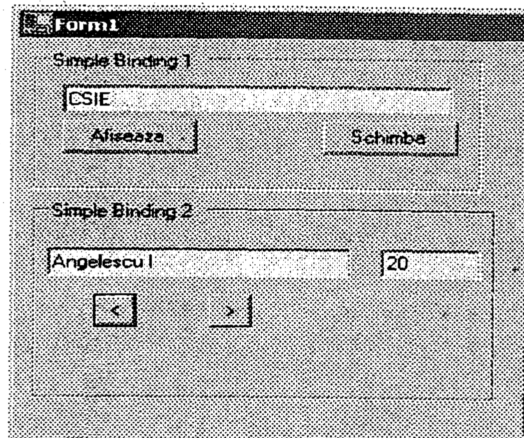
Cum identificarea legăturii se face pornind de la sursa de date (`vectPers`), aceste instrucțiuni realizează defilarea în vector și actualizarea concomitentă **a tuturor controalelor** care sunt legate de acest vector (în cazul nostru, și numele și vârsta, din cele două textBox-uri).
Sintetizând, putem spune că pentru gestiunea unei legături simple, NET Framework lucrează cu două tipuri de obiecte:

- câte un **CurrencyManager**, pentru fiecare sursă de date legată la o interfață;
- un obiect **BindingContext**, care ține evidența tuturor obiectelor de tip **CurrencyManager.**

BindingContext -ul dispune de o **indexare**, care dacă primește numele unei surse de date, returnează obiectul **CurrencyManager** care se ocupă de legarea ei la un control, sau la mai multe, de pe o formă.
Instrucțiunea:

```
BindingContext[vectPers].Position+=1;
```

trebuie interpretată în sensul următor: **CurrencyManager**-ul gestionează legăturile vectorului `vectPers` cu controale existente pe formă; în acest scop, el păstrează și poziția curentă în vector (cea care este preluată de controale, la un moment dat); **BindingContext**-ul primește numele unei surse de date și returnează prin indexare instanța **CurrencyManager** aferentă acelei surse; la rândul lui obiectul **CurrencyManager** își modifică poziția curentă în vector, făcând să se preia alt set de valori (nume, vârstă) în controalele de afișare text. Funcția poate fi redactată și altfel, detaliind instrucțiunea `BindingContext[vectPers].Position+=1;`:




```
private void btnStanga_Click
                        (object sender, System.EventArgs e)
{
    CurrencyManager crtMngr;
            // explicitare lucru cu CurrencyManager
    crtMngr=(CurrencyManager)BindingContext[vectPers];
    crtMngr.Position-=1;
}
```

Pot fi legate toate clasele derivate din `Windows.Forms.Control`, cu oricare din proprietățile lor. Practic, legarea simplă oferă un mecanism foarte practic pentru a construi interfețe grafice, care se reconfigurează în funcție de diverse date.

Exemple de utilizare a legării datelor.

1. Afișarea unui set de fotografii, legând un vector de imagini de proprietatea Image a unui control PictureBox.
2. Corelarea automată a proprietății Value a unui control ProgressBar cu magnitudinea relativă a observațiilor despre un fenomen.
3. Corelarea culorii unui panel cu valorile unui decodor de culori.

Controalele de tip container pot avea la rândul lor propriul **BindingContext**, care gestionează manageri de legături pentru controalele pe care acestea le dețin și care pot lega aceleași sau alte surse de date.

Sursele de date care au și un corespondent vizual în Designer, pot fi văzute de acesta, iar legarea se poate face și vizual, selectând controlul, **Properties / Data Bindings / Advanced** unde se aleg atât proprietatea ce se va lega, cât și sursa de date.

2. Legarea complexă - complex data binding

Simple Binding conecta un singur element dintr-un model de date (un element din vector, în cazul nostru) de o singură proprietate a unui control. **Complex Binding** conectează un control cu întreaga colecție de date a unei surse. Spre exemplu, un grid poate afișa dintrodată, o întreagă tabelă dintr-o bază de date, cu toate câmpurile ei.

Complex Binding reprezintă o modalitate comodă de a conecta volume mari de date cu elementele de interfaţă cu utilizatorul. Nu însă toate controalele pot beneficia de o astfel de legare, ci doar controalele orientate pe colecţii, cum ar fi: ComboBox, ListBox, Grid etc.

Ca exerciţiu, în aplicaţia anterioară vom adăuga un ComboBox, iar pe un buton inscripţionat "Leaga" adăugăm funcţia de tratare Click conţinând doar liniile de cod:

```
comboBox1.DataSource = vectPers;
comboBox1.DisplayMember = "Nume";
comboBox1.ValueMember="Varsta";
```

La rulare, apăsînd butonul "Leaga" constatăm popularea casetei combinate cu persoanele din vectorul de persoane; mai mult decât atât, la selectarea unei persoane, constatăm că se modifică şi persoana afişată în TextBox-ul `txtPers`, **de la legătura simplă**. Acest lucru se explică prin faptul că acelaşi BindingContext gestionează ambele conexiuni cu ajutorul aceluiaşi obiect **CurrencyManager**, iar modificând poziţia curentă pe o sursă de date, toate conexiunile existente cu această sursă se aliniază la noua poziţie curentă !

Lucrurile sunt valabile şi reciproc: defilând în vector cu ajutorul celor două butoane de navigare ale legăturii simple, se modifică şi persoana afişată în zona de editare din ComboBox.

Exerciţiu

Editaţi numele persoanelor din **txtPers** şi defilând, constataţi ce va afişa ComboBox-ul. **Invers**, editaţi numele persoanei selectată în zona de editare din ComboBox şi observaţi dacă legătura simplă sesizează astfel de modificări. Trataţi un eveniment la nivelul casetei combinate astfel încât să-şi modifice efectiv datele din zona listă (nu din vector !) atunci când ele au fost editate şi constataţi acum efectul modificării lor asupra vectorului şi asupra celorlalte controale legate la vector.

Rezolvare

Compus din două părţi (zone de editare şi zona listă), comboBox-ul necesită **actualizarea explicită** a zonei listă, cu valorile modificate ale vectorului.

Afişarea unui vector în grid

Există controale specializate în afişarea simultană a mai multor date, cu mai multe atribute. Spre exemplu, informaţiile despre mai multe persoane, cu toate toate **proprietăţile** definite în clasă (Nume şi Varsta), pot fi afişate extrem de simplu, într-un grid:

- se declară sau se trage din ToolBox un obiect DataGrid;

```
private DataGrid dataGrid1;
```

- pe un eveniment, se declară vectorul ca sursă de date care alimentează gridul:

```
dataGrid1.DataSource = vectPers;
```

Gridul se configurează automat şi va afişa atâtea coloane, câte **proprietăţi** sunt definite în clasa `Pers`. De observat că legarea complexă necesită doar fixarea proprietăţii DataSource a controlului, nu adăugare în colecţia de legături.

În fereastra de **Properties** asociată controlului de tip grid, în partea de jos există un link spre o fereastră secundară de dialog pentru a stabili un `AutoFormat`. Pentru a stabili caracteristici ce nu apar în şabloanele oferite în fereastra de `AutoFormat`, se poate apela la proprietăţile gridului afişate în Properties, modificându-le static sau dinamic, prin cod sursă.

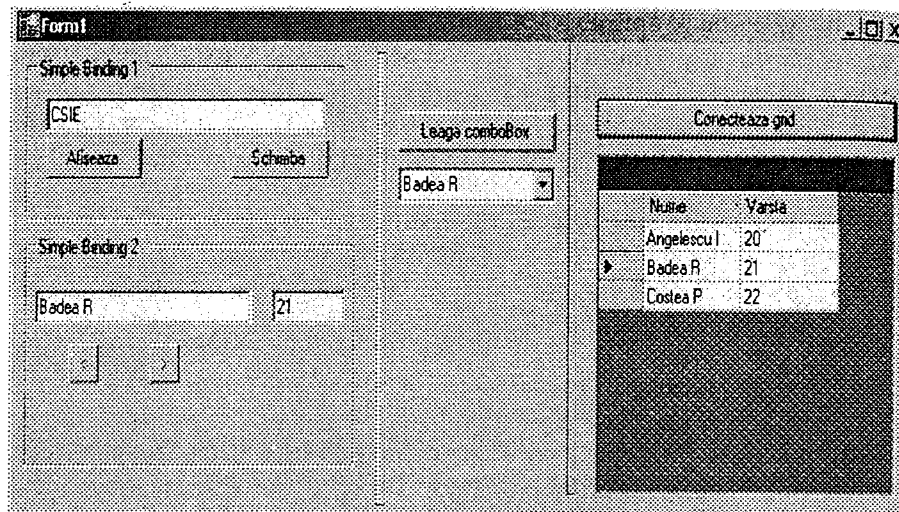
Modificând datele din grid, observăm că şi conţinutul vectorului de persoane va fi modificat; toate legările simple şi complexe operate anterior, confirmă şi ele modificările făcute.

Se poate vorbi aşadar de:

- **legare unidirecţională**, când modificările din sursa de date sunt operate şi în controale, dar modificările din controale nu au efect asupra datelor din sursă; este cazul legării realizate prin ComboBox; modificarea numelui din zona de editare a casetei nu afectează numele stocate în vector.

- **legare bidirecţională**, când modificările din sursa de date sunt vizibile în controale, iar modificările operate în controale se răsfrâng şi asupra datelor din sursa de date; este cazul legării simple

prin intermediul textbox-ului și a legării complexe realizate prin intermediul gridului.



ADO.NET – OBIECTELE DE LUCRU CU BAZE DE DATE

1. **Introducere în tehnologia ADO.NET**
2. **Conexiunea la baza de date**
3. **DataAdapter - adaptorul**
4. **DataSet – mulțimea de date**
5. **Tabela de date – DataTable**
6. **Relațiile dintre date – obiectul DataRelation**
7. **DataRow – tuplu de date**
8. **DataView – vizualizarea**
9. **Obiecte de tip Command**
10. **Lucru cu procedurile stocate**
11. **Typed și untyped DataSet**

1. Introducere în tehnologia ADO.NET

În principiu, accesul la comenzi SQL dintr-un limbaj de programare se poate face prin mai multe modalități dintre care cele mai frecvente sunt:

- **ODBC (Open Data Base Connectivity)**, o interfață standardizată care transformă o cerere SQL în apeluri de funcții acceptate de un driver ODBC, care recunoaște mecanismul de stocare și regăsire a datelor, specific unui SGBD;
- prin **OLEDB (Object Linking and Embedding**), un set de obiecte specilizate în stocarea și regăsirea datelor și care expun:
  - **consumer interface** : o interfață de nivel înalt, care oferă accesul dintr-un limbaj de programare la comenzi SQL ;
  - **provider interface**, care grupează apelurile de nivel jos, specifice unui SGBD.

Prima categorie de apeluri este transormată în apeluri de pe nivelul provider; pentru că nivelul consumer era totuși prea jos, s-a creat **ADO - Active Data Object**, ca un alt nivel, și mai înalt de abstractizare.

ADO.NET este o dezvoltare a acestei tehnologii de acces la date. Ea disponibilizează **clase, interfețe, structuri și enumerări** pentru accesul la baze date sub **.NET Framework**. Este proiectată să lucreze și deconectat de la baza de date, stocând datele în memoria ***cache*** locală.

Datele stocate pot fi modificate, iar modificările sunt operate abia a sfârșit, în BD. Avantajul vitezei de lucru este afectat de faptul că în lucrul pe

BD partajate nu avem cele mai recente actualizări, motiv pentru care această tehnologie nu este eficientă pentru aplicațiile real-time, gen brokeraj. Ca urmare, ADO.NET include şi un mecanism de acces direct la BD prin intermediul unor obiecte `DataReader`.

O altă caracteristică este folosirea ca suport alternativ a limbajului XML, făcând astfel o legătură între **modelul relațional** de lucru cu BD şi modelul **ierarhic**, adoptat şi în accesul la baze de date plasate la distanță, în Internet, model descris cu uşurință în limbajul XML.

ADO.NET disponibilizează prin clasele sale:

- obiecte pentru conectare la sursa de date;
- obiecte ce gestionează comenzi de manipulare;
- obiecte container de date.

In plus, **ADO.NET expune şi notifică producerea unor evenimente,** permițând programelor să lucreze altceva în paralel sau să se alinieze şi să ia decizii în funcție de aceste evenimente.

Ca o **observație** pentru programatorii sub ADO MFC, obiectul **RecordSet** din MFC este înlocuit în .NET cu două obiecte: un `DataAdapter` şi un `DataSet`, care permit operații pe baza de date chiar deconectați (pentru acces rapid) şi din mai multe aplicații simultan.

Sub Visual Studio .NET, există două abordări posibile pentru realizarea de aplicații cu baze de date:

- **prin program,** scriind direct cod sursă ce instanțiază şi foloseşte obiecte specifice lucrului cu baze de date;
- **vizual,** folosind controalele disponibilizate de Visual C# şi asociate (categoria **Data**, din ToolBox) unor surse de date.

Vizual se realizează şi se testează uşor programele cu un înalt grad de interactivitate, în timp ce pentru programe cu grad scăzut de interactivitate, cum ar fi elaborarea rapoartelor periodice, se preferă forma „programatică", scriind cod sursă care foloseşte valori prestabilite, în locul celor introduse de utilizator.

În demersul nostru, preferăm inițierea în ADO.NET folosind prima alternativă, pentru a înțelege exact ce obiecte şi ce metode există şi cum cooperează ele, după care se abordează şi lucru vizual cu baze de date,

punând Designer-ul să scrie cod şi exploatând interfața vizuală de acces la baze de date, programatorul doar completând, în cunoştiință de cauză, acest cod.

Obiectele cele mai importante prin care se implementează accesul la o bază de date sunt: **conexiunea, adaptorul** şi **setul de date.**

2. Conexiunea la baza de date

Este responsabilă cu realizarea legăturii fizice dintre o bază de date şi aplicația .NET şi depinde de tipul furnizorului de date. Diferențele de implementare de la un furnizor la altul sunt majore, astfel încât generalizări se pot obține doar folosind tehnici de legare şi încapsulare obiecte (OLE) sau drivere Open Data Base Connectivity (ODBC).

Furnizorii frecvenți de surse de date (data providers) sunt:

- SQL Server .NET Data Provider pentru **SQL Server 7.0**, scris complet în *managed code* şi considerat modul nativ de acces la BD sub ADO.NET
- **OLE DB** .NET Data Provider pentru SQL Server 6.5 sau versiuni anterioare, Oracle şi Microsoft Access.
- **ODBC** .NET pentru a răspunde dezvoltatorilor de aplicații care utilizează drivere ODBC pentru acces la date.

O conexiune se poate crea în mai multe moduri:

1. vizual, folosind Server Explorer, din mediul integrat;
2. prin program, folosind obiecte de tip **`SqlConnection`**, **`OleDbConnection`**, sau **`OdbcConnection`**;
3. implicit, cerând unui `DataAdaptor` pe metoda **`Fill( )`** să încarce date dintr-o bază de date.

Iată cum se descriu cele trei tipuri de conexiuni, specifice fiecărui provider.

- conexiune OLEDB:

```
string sirConex =
  "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\\prod.mdb";
OleDbConnection myConex = new OleDbConnection(sirConex);
```

- conexiune SQL

```
string connectString =
      "server=localhost;database = BDClienti";
```

```
SqlConnection conn = new SqlConnection(connectString);
/* ... */
conn.Open();
/* ... */
if (conn.State == ConnectionState.Open) conn.Close();
```

- conexiune ODBC

```
OdbcConnection conODBC; //  există using System.Data.Odbc;
System.Data.SqlClient.SqlConnection conSQL;
                        // namespace dat explicit
```

OleDbConnection este definită în namespace-ul **System.Data.OleDB**, **SqlConnection** în **System.Data.SqlClient**, iar **OdbcConnection** în **System.Data.Odbc**.

Stările unei conexiuni şi semnificaţiilor lor sunt următoarele:

- **Connecting** – în curs de conectare, conexiune nedeschisă încă;
- **Open** – conexiune deschisă;
- **Executing** – în derularea unei comenzi;
- **Fetching** – în timpul unei căutări în BD

Într-o aplicaţie simplă, care declară `using System.Data.OleDb` se poate pune pe evenimentul click al unui buton codul:

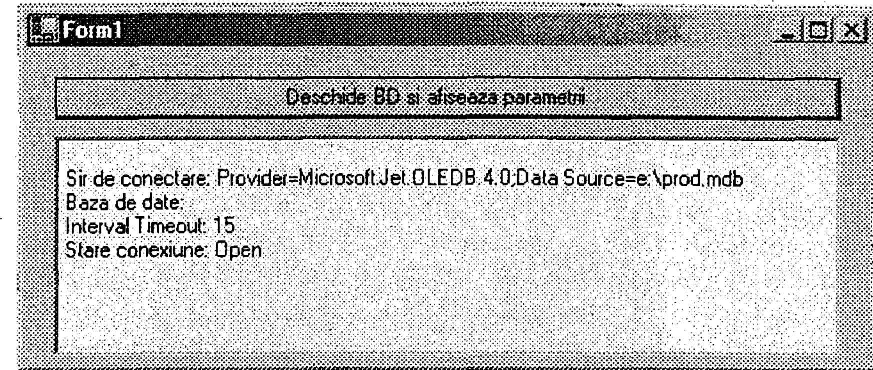
```
private void button1_Click(object sender, System.EventArgs e)
{
 string sirConex =
 "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\\prod.mdb";
 OleDbConnection conex=new OleDbConnection(sirConex);
  try
    {
          conex.Open();
    }
  catch { MessageBox.Show("Esuare la deschidere BD"); }

  if(conex!=null)
      {
       textBox1.Text="\r\n Sir de conectare: "+
                 conex.ConnectionString;
       textBox1.Text+="\r\n Baza de date: "+
                 conex.Database;
       textBox1.Text+="\r\n Interval Timeout: "+
                   conex.ConnectionTimeout.ToString();
       textBox1.Text+="\r\n Stare conexiune: "+
```



```
                   conex.State.ToString();
      }
  conex.Close();
}
```

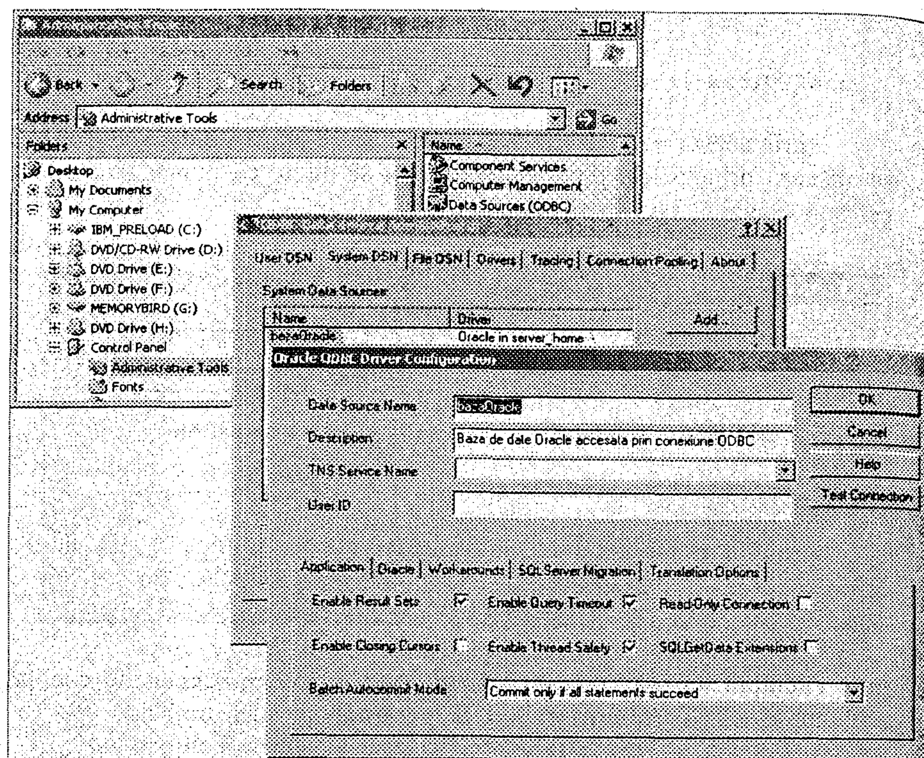
Prin acest cod se deschide o conexiune de tip **OleDbConnection** şi i se afişează principalele proprietăţi. Ele pot să difere ca număr şi conţinut pentru o conexiune de tip **SqlConnection.**



Spre exemplu, un string de conectare pentru `SqlConnection` nu conţine atributul `Provider` (Microsoft SQL Server fiind provider implicit).

Un string de conectare pentru ODBC lucrează cu **Data Source Name** (DSN) înregistrat în prealabil în **Registry**. Pentru aceasta, sub sistemul de operare se alege **Settings / Control Panel / Administrative Tools / Data Sources (ODBC)**; în funcţie de drivere-le instalate în sistem se optează apoi pentru **adăugarea**, **eliminarea** sau **reconfigurarea** unei surse de date.

Conexiunile se pot crea şi vizual, folosind Server Explorer. Sub Visual Studio, Server Explorer-ul permite la momentul proiectării vizuale a aplicaţiei, să adăugăm noi conexiuni sau să ştergem conexiuni existente, să deschidem sau să închidem conexiuni, să gestionăm cozi de mesaje sau evenimente de conectare la o bază de date. Detaliile privind aceste facilităţi sunt prezentate în capitolul **Programarea vizuală a lucrului cu baze de date.**



3. DataAdapter - adaptorul

Obiectul `DataAdapter` mediază schimburile de date dintre un obiect `DataSet` și baza de date, atât pentru partea de regăsire și încărcare a datelor, cât și pentru salvarea datelor modificate. El ține:

- un **set de comenzi de manipulare** a datelor din BD
- **conexiunea la BD** folosită la popularea DataSet-ului.

Mai precis, adaptorul asociază o conexiune de cele patru obiecte de tip comenzi, ce pot fi executate pe o bază de date: **`SelectCommand`, `InsertCommand`, `UpdateCommand`** și **`DeleteCommand`**.

Principala sa metoda, **`Fill( dataSet, tabela)`** încarcă cu date o tabelă dintr-un DataSet. În funcție de comanda `Select` a adaptorului, o tabelă din DataSet poate conține date provenind din mai multe tabele din baza de date.

DataAdapter-ul lucrează cu conexiuni deschise sau închise, caz în care deschide și închide el implicit conexiunea. Dacă însă am deschis noi explicit o conexiune cu `Open()`, atunci este important să închidem tot explicit conexiunea, după ce am folosit-o (adică după ce am apelat metoda **`Fill`** a DataAdapter-ului), căci DataAdapter-ul nu o va închide implicit.

Comenzile conținute de un adaptor sunt expuse ca proprietăți, fiind la rândul lor obiecte de tip **`SqlCommand`** sau **`OleDbCommand`.**

Fiecare **DataAdapter** mediază transferul de date între un singur obiect **DataTable** din **DataSet** și rezultatul unei singure interogări printr-o comandă SQL; deci dacă sunt mai multe tabele sau mai multe tipuri de intrerogări se vor folosi mai multe obiecte **DataAdapter,** câte unul pentru fiecare interogare, **sau se redefinesc de fiecare dată proprietățile adaptorului** (obiectele de tip ***XxxCommand***), pentru a corespunde fiecărui caz în parte.

Proprietatea **`SelectCommand`** folosește la precizarea acțiunii de executat pentru umplerea DataSet-ului cu datele dintr-o bază de date. Mai precis, ea conține fraza select necesară construirii metodei `Fill` a adaptorului de date, metodă ce suprascrie metoda `Fill` moștenită din clasa de bază și cerută de interfața `IDataAdapter`.

`InsertCommand`, `UpdateCommand` și **`DeleteCommand`** sunt apelate pentru a actualiza datele **și în baza de date**, când utilizatorul a efectuat inserări, modificări și respectiv ștergeri, asupra DataSet-ului. Aceste comenzi stau la baza execuției metodei `Update()`.

Metoda **`Update()`** a DataAdapter-ului este sintactic asemănătoare metodei `Fill()`; ambele primesc ca parametri un obiect DataSet și un obiect DataTable; similitudinea se explică prin faptul că au funcții comparabile, dar de sens opus: `Fill()` citește datele din baza de date în dataSet, `Update()` scrie datele din DataSet în baza de date.

Comenzile pot fi generate și automat, doar **`SelectCommand`** fiind obligatorie; ea poate fi scrisă direct ca proprietate sau poate fi dată ca parametru în constructorul DataAdaptor-ului. Celelalte comenzi trebuie să existe doar în momentul în care se invocă metoda **`Update()`** expusă de un adaptor și în DataSet există tipul de modificare ce impune comanda respectivă.

Exercițiu

Să se testeze lucru cu un adaptor.

Rezolvare

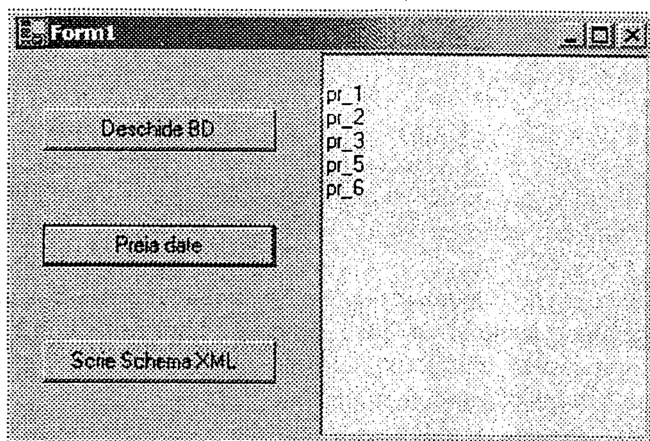
Aplicație simplă, Visual C# care anunță:

```
using System.Data.OleDb;
using System.Data.SqlClient;
```

La nivelul formei se declară un obiect DataSet și un obiect DataAdapter:

```
private     DataSet dsProd ;
private OleDbDataAdapter daProd;
```

Vizual se adaugă trei butoane și un textBox, docat `Left`.



Pe butonul **`Deschide`** se pune funcția de tratare Click:

```
private void Deschide_Click
                         (object sender, System.EventArgs e)
{
  // dsProd si daProd sunt variabile globale in forma
  dsProd = new DataSet("dsProduse");

  string frazaSQL = "select codp, denum, pret from produse";
  string sirConex =
 "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\\prod.mdb";

  daProd = new OleDbDataAdapter(frazaSQL, sirConex);
```



```
  // variabile locale in functie
  OleDbDataAdapter daAdapter2 = new OleDbDataAdapter
              ("select codm,denm from materiale", sirConex);

  OleDbDataAdapter daAdapter3 = new OleDbDataAdapter
                   ("select * from consumuri", sirConex);
}
```

Variabila **`dsProd`** de tip `DataSet`, declarată la nivelul formei, este aici instanțiată, obiectul primind și un nume, în afara referinței.

Un adaptor **`daProd`**, declarat global la nivelul formei, este instanțiat pornind de la o frază select și de la un șir de conectare, fiind folosit în altă funcție la popularea DataSet-ului, cu date preluate din tabela *produse*.

După cum se observă, uneori tabelele sunt asociate implicit DataSet-ului, prin intermediul adaptorului, care citează tabela în fraza select; în acest caz nu mai este nevoie să adăugăm manual tabele la colecția `Tables` a DataSet-ului.

Funcția de mai sus declară și alte variabile locale de tip **`OleDbDataAdapter`**, care stochează comenzi proprii de interogare și care pot fi folosite **doar temporar și numai în această funcție**.

Pe butonul **`Incarca`** tratăm click de mouse cu funcția:

```
private void Incarca_Click(object sender, System.EventArgs e)
{
      daProd.Fill(dsProd);
      // deschide, umple dataSet si inchide automat conexiune

      foreach(DataRow lin in dsProd.Tables[0].Rows)
           textBox1.Text+= "\r\n"+lin["denum"];
}
```

Codul de mai sus, prin intermediul obiectului `daProd` de tip `OleDbDataAdapter`, încarcă date în DataSet-ul `dsProd`, instanțiat în funcția anterioară.

Metoda **`Fill`** folosește la **adăugarea** unor linii sau împrospătarea liniilor de date preluate din baza de date și stocate în tabela unui DataSet; dacă nu se dorește adăugarea la DataSet-ul deja încărcat o dată, acesta trebuie mai întâi golit, invocându-i-se metoda **`Clear`**`()`.

Secvența de mai sus poate fi pusă pe evenimentul `Click` al unui buton, într-o aplicație Windows Forms, care declară și `using System.Data.OleDb;`

Liniile de date rezultate din interogare sunt apoi iterate pentru a obține un mesaj cu denumirea produselor.

Consultând baza de date, dataAdapter-ul află și schema de descriere a tabelei asociate. Această schemă poate fi salvată distinct și consultată sau folosită ulterior.

Pe butonul **ScrieXML** se adaugă codul pentru salvarea schemei de descriere a tabelei *produse*:

```
private void ScrieXML_Click
                              (object sender, System.EventArgs e)
{
      dsProd.WriteXmlSchema("produse.xsd");
}
```

Se poate vizualiza cu Wordpad fișierul **produse.xsd** și apoi se poate da click în Windows Explorer pe fișierul respectiv, fiind vizualizată schema direct în editorul de sub Visual, unde schema poate fi și modificată vizual.

Deși e o metodă de-a DataSet-ului, **WriteXmlSchema** scrie scheme însușite prin intermediul adaptoarelor, care au consultat baza de date și au extras schemele de descriere ale tabelelor asociate.

Fișierul XSD este un fișier XML de tip special, în format extern, care poate conține atât datele propriu-zise, cât și metadatele (descrierea datelor) pentru a le putea interpreta corect la destinație.

De asemenea, fișierul **produse.xsd** poate fi folosit sub un sistem de gestiune a bazelor de date, **la importul unei tabele din exterior** (de exemplu, sub Access, cu o bază de date deschisă, **File / Get External Data / Import** selectând fișiere de tip ***.xsd** se poate face un import de tabelă) .

4. DataSet – mulțimea de date

Obiectul **DataSet** este containerul de date; el corespunde unei implementări relaționale a unei baze de date în memorie. Când datele provin din mai multe tabele se recomandă definirea câte unui adaptor pentru fiecare tabelă.

Structura unui obiect DataSet se prezintă în fig. 14.1

Când DataSet-ul conține mai multe tabele, fiecare cu câte un adaptor de date asociat, trebuie invocată metoda **Fill** a fiecărui adaptor.

Obiectul DataSet marchează informația care a fost modificată (la nivelul fiecărei linii de date există proprietatea **RowState**) și menține o copie și a informației originale, astfel încât o putem restaura când acest lucru se impune.

La apelul metodei **Update()**, DataAdapter-ul analizează dacă sunt modificări în DataSet și lansează **InsertCommand**, **UpdateCommand**, ori **DeleteCommand**, în funcție de tipul actualizărilor produse : adăugare de noi înregistrări, modificări sau ștergeri.

Comenzile **InsertCommand**, **UpdateCommand**, ori **DeleteCommand** trebuie să existe scrise înainte de apelul metodei **Update**, altminteri este semnalată o excepție, ori de câte ori lipsește una din comenzile de actualizare și ea este necesară, deoarece există actualizări de acel tip.

Corespunzând unei reprezentări a datelor în memorie, DataSet-ul nu știe de unde provin datele pe care le conține, ci cunoaște doar structura lor relațională. Ca urmare după o actualizare a bazei de date cu modificările operate în DataSet, trebuie notificată acceptarea sau refuzul de către baza de date, de a opera modificările făcute; această notificare se face printr-un apel al metodelor **AcceptChanges** sau **RejectChanges**; în primul caz se face automat un "commit", iar în al doilea caz se va face automat un "roll-back" pentru a aduce DataSet-ul la starea de dinaintea update-ului.

DataSet cu tabele multiple

Așa cum am văzut deja, un **DataSet** poate conține mai multe obiecte `DataTable`, câte unul pentru fiecare tabelă din baza de date. Mai există însă o nuanță: printr-o comandă `Select` bazată pe un ***join*** putem aduce date provenind din **mai multe tabele** ale bazei de date **într-un singur obiect DataTable** al unui **DataSet**.

Dacă se dorește actualizarea simultană a datelor provenind din mai multe tabele, trebuie specificat acest lucru explicit prin comenzile de update, deoarece **dataSet**-ul lucrând independent de baza de date, toate relațiile care au stat la baza încărcării inițiale, nu mai sunt cunoscute ulterior.

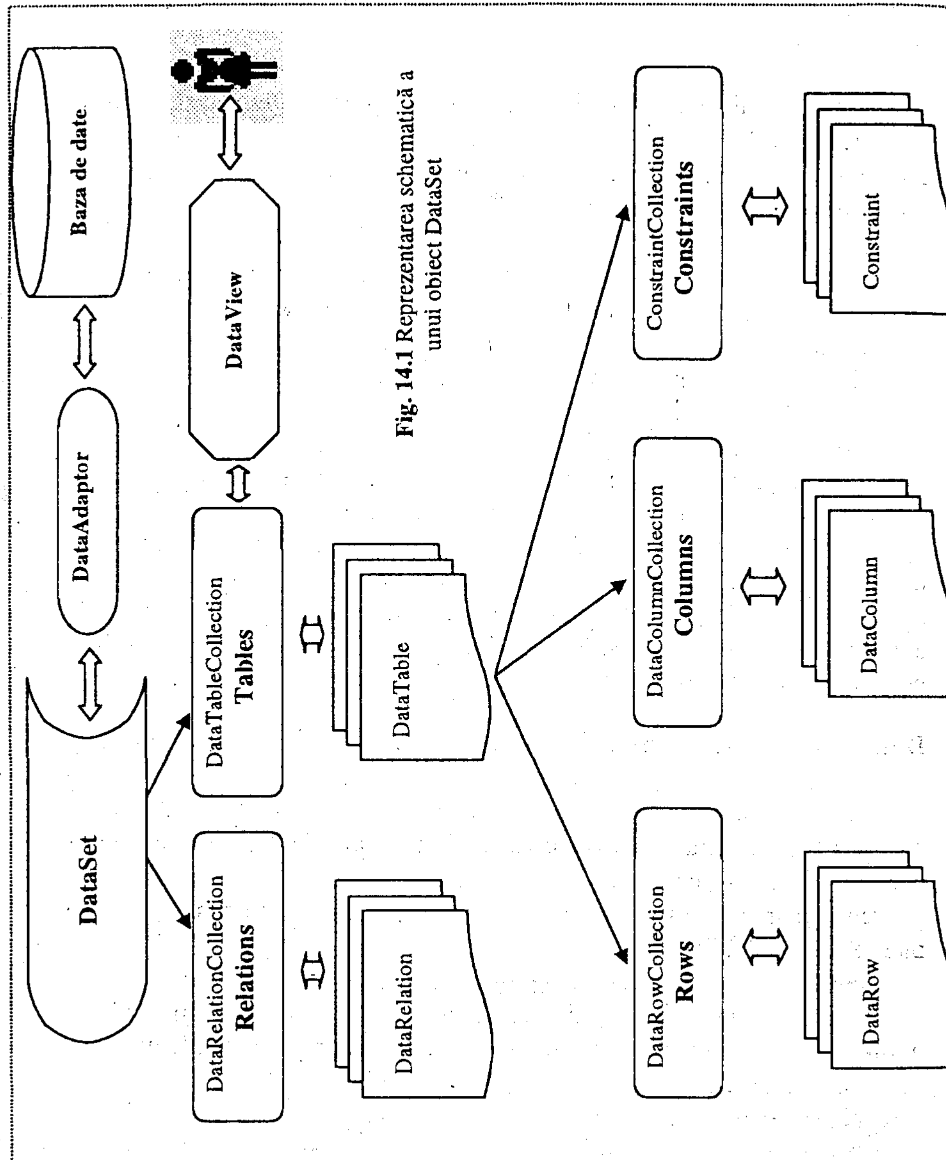


Fig. 14.1 Reprezentarea schematică a unui obiect DataSet

5. Tabela de date – DataTable

- se poate crea independent sau legată de un DataSet.

```
DataTable t1 = new DataTable() ;
      // primeste nume implicit Table1
DataTable t2 = new DataTable("tabela 2") ;
      // primeste nume dat de programator
```

- legarea de un DataSet, prin adăugarea tabelei în colecţia Tables:

```
DataSet ds = new DataSet();
ds.Tables.Add(t1); ds.Tables.Add(t2);
ds.Tables.Add();
// adaugarea unei tabele de nume implicit Table2
foreach(DataTable crt in ds.Tables)
            msg+="\n"+crt.TableName;
MessageBox.Show(msg);
```

Exerciţiu

Să se exemplifice construirea, încărcarea şi exploatarea unui DataSet conţinând două tabele, **Clienţi** şi **Comenzi**. Toate operaţiunile se execută doar în memorie, fără vreo conexiune cu o bază de date propriu-zisă. Se va defini o relaţie între datele conţinute în cele două tabele, astfel încât să poată fi regăsite toate comenzile aferente unui client, indicat prin cod client.

Rezolvare

Pe o aplicaţie Windows se pun:

- un buton care să declanşeze toate acţiunile derulate în cadrul aplicaţiei
- un textBox în care se introduce codul clientului ale cărui comenzi sunt afişate şi totalizate
- un textBox ce va afişa datele despre client (cod client şi nume)
- un listBox ce afişează comenzile câte unui client şi total cantitate comandată
- `public DataSet ds;` dată membră ce referă setul de date cu care se lucrează în mai multe funcţii ale aplicaţiei.

Pentru declararea şi încărcarea dataSet-ului se scrie o funcţie `CreazaSet`, ce va fi apelată din constructorul formei aplicaţiei:

```
private void CreazaSet()
{
	ds = new DataSet("myDataSet");

	// Creaza tabela Clienti
	DataTable tClienti = new DataTable("Clienti");

	// Creaza si adauga doua coloane la tabela Clienti
	DataColumn CodClientClienti =
		new DataColumn("CodClient", typeof(int));
	DataColumn cNumeClient = new DataColumn("NumeClient");
	tClienti.Columns.Add(CodClientClienti);
	tClienti.Columns.Add(cNumeClient);

	// Creaza tabela Comenzi
	DataTable tComenzi = new DataTable("Comenzi");

	// Creaza si adauga trei coloane la tabela Comenzi
	DataColumn cCod =
		new DataColumn("CodClient", typeof(int));
	DataColumn cData_Comenzii =
		new DataColumn("Data_Comenzii",typeof(DateTime));
	DataColumn cCantitate =
		new DataColumn("Cantitate", typeof(int));
	tComenzi.Columns.Add(cCod);
	tComenzi.Columns.Add(cCantitate);
	tComenzi.Columns.Add(cData_Comenzii);

	// Leaga tabelele la DataSet.
	ds.Tables.Add(tClienti);	ds.Tables.Add(tComenzi);

	// Creaza o relatie intre tabele, mediata de CodClient
	// DataRelation dr = new DataRelation
	//	("Client_Comenzi", CodClientClienti , cCod);
	DataRelation dr =
		new DataRelation("Client_Comenzi",
			tClienti.Columns[0] ,tComenzi.Columns[0]);
	ds.Relations.Add(dr);

	// Populare tabele
	DataRow linClienti; DataRow linComenzi;

	// trei Clienti in tabela.
	for(int i = 1; i < 4; i++)
	{
		linClienti = tClienti.NewRow();
		linClienti["CodClient"] = i;
		linClienti["NumeClient"] = "Client_"+i;
		tClienti.Rows.Add(linClienti);
	}
```



```
	// Creare Comenzi pentru fiecare client.
	for(int i = 1; i < 4; i++)
	{
		for(int j = 1; j < 6; j++)
		{
			linComenzi = tComenzi.NewRow();
			linComenzi["CodClient"]= i;
			linComenzi["Data_Comenzii"]=
				new DateTime(2004, i, j * 2);
			linComenzi["Cantitate"] = i * 10 + j * 1;
			// Adauga linia la tabela Comenzi
			tComenzi.Rows.Add(linComenzi);
		}
	}
}
```

Funcția declară un obiect DataSet numit **myDataSet** (a nu se confunda numele dataSet-ului cu referința `ds`) și două obiecte DataTable ce referă două tabele numite `Clienti` și `Comenzi`. Lor li se constituie colecția **Columns** prin adăugarea de câmpuri ce definesc coloanele fiecărei tabele (`CodClient`, `NumeClient`, pentru prima tabelă, respectiv `CodClient`, `Data_Comenzii` și `Cantitate`, pentru a doua tabelă). Nu trebuie confundate **referințele** de `DataColumn` cu **denumirile** de coloane, care sunt stocate ulterior și în baza de date; referințele de obiecte sunt folosite doar temporar la adăugarea coloanelor la colecția **Columns**, a DataSet-ului.

Tabelele astfel configurate sunt adăugate apoi la colecția **Tables** a DataSet-ului.

6. Relațiile dintre date – obiectul DataRelation

DataRelation este un obiect care înregistrează legăturile (relațiile) dintre tabele și permite explorarea lor pentru regăsirea datelor.

Spre exemplu, o legătură **1-M** între tabela referită prin `tClienti` și tabela referită prin `tComenzi` permite regăsirea facilă a tuturor comenzilor făcute de un client.

Dispunând de cele două tabele deja atașate unui DataSet, putem introduce o relație între tuplurile celor două coloane, dând un nume relației și precizând cele două coloane, "parent" și "child", aparținând celor două tabele diferite, care mediază legătura.

- când cunoaștem **referința** fiecărei DataColumn, sub forma :

```
DataRelation dr =
   new DataRelation("Client_Comenzi", CodClientClient, cCod);
ds.Relations.Add(dr);
```

- când cunoaştem **poziţia** coloanelor în colecţia Columns a fiecărei tabele:

```
DataRelation dr = new DataRelation("Client_Comenzi",
                    tClienti.Columns[0] ,tComenzi.Columns[0]);
ds.Relations.Add(dr);
```

- când cunoaştem **numele** coloanelor ce mediază relaţia dintre cele două tabele:

```
DataRelation dr = new DataRelation("Client_Comenzi",
                          tClienti.Columns["CodClient"],
                          tComenzi.Columns["CodClient"]);
ds.Relations.Add(dr);
```

Urmează apoi **popularea celor două tabele** cu date generate ad-hoc; pentru încărcare este nevoie de obiecte `DataRow`, care referă câte o linie de date din fiecare tabelă, linia fiind structurată conform structurării pe coloane a fiecărei tabele; acest lucru face ca obiectele `DataRow` să nu aibă constructori proprii, ci să fie construite uzual folosind metoda `NewRow()` ce aparţine unei tabele, care îşi cunoaşte propria colecţie de coloane.

`DataRow` crează în acelaşi timp şi containerul de date, astfel încât refolosim referinţa `linClienti`, dar dăm `tClienti.NewRow()` în `for`, creând câte un container distinct pentru fiecare tuplu de date din tabelă.

Un element din linia de date este referit prin numele coloanei căreia aparţine sau prin indexul poziţiei pe care o ocupă (`NumeClient` este a doua coloană, deci ocupă poziţia 1):

```
linClienti["NumeClient"] = "Client_"+i;
```

sau

```
linClienti[1] = "Client_"+i;
```

Funcţia care tratează evenimentul `Click` pe butonul de căutare a comenzilor aferente unui client are următorul conţinut:

```
private void cauta(object sender, System.EventArgs e)
{
      int cc = Convert.ToInt32(eCod.Text);
                                  // preluare cod client
      Decimal Total = 0;        int cant = 0;
```



```
      DataTable tClienti = ds.Tables["Clienti"];

      DataColumn[] vect_chei = new DataColumn[1];
      vect_chei[0] =
                ds.Tables["Clienti"].Columns["CodClient"];
      ds.Tables["Clienti"].PrimaryKey = vect_chei;

      DataRow linSelectata = tClienti.Rows.Find(cc);
      //DataRow linSelectata = tClienti.Rows[0];

      //Navigheaza spre comenzi folosind relatia Client_Comenzi
      DataRow[] vectLinii =
                linSelectata.GetChildRows("Client_Comenzi");
      // Afiseaza Clientul
      afis.Text+=(cc+" "+linSelectata["NumeClient"]+".   " );
      string linia="";
      for(int i = 0; i < vectLinii.Length; i++)
      {
       linia="";
       foreach(DataColumn colCrt in ds.Tables["Comenzi"].Columns)
                 linia+=vectLinii[i][colCrt]+"   ";
       listBox1.Items.Add(linia);
       cant = (int)vectLinii[i]["Cantitate"];
       Total += cant;
      }
      linia=""; linia+="----------------------";
      listBox1.Items.Add(linia);
      linia=""; linia+=Total; listBox1.Items.Add(linia);
}
```

După ce extrage codul introdus de utilizator pentru clientul ce face obiectul căutării, funcţia pregăteşte căutarea după o cheie; în acest sens se înregistrează câmpul `CodClient` din tabela `Clienti`, drept cheie primară.

```
DataColumn[] vect_chei = new DataColumn[1];
vect_chei[0] = ds.Tables["Clienti"].Columns["CodClient"];
ds.Tables["Clienti"].PrimaryKey = vect_chei;
```

Pe scurt, cheia primară se înregistrează setând proprietatea `PrimaryKey` a tabelei.

O cheie primară poate fi compusă din mai multe subchei; ca atare se declară un vector de `DataColumn`, ce pot reprezenta subchei, iar la instanţierea vectorului se precizează din câte câmpuri se compune (unul, în exemplul nostru); se încarcă apoi fiecare element din vector, cu referinţa câte unei coloane care intră în componenţa cheii primare.

În final, referința vectorului conținând cheia primară se înregistrează drept proprietate **PrimaryKey** a tabelei `Clienti`. O dată înregistrată o cheie primară, putem folosi căutarea după cheie, apelând la metoda **Find** a colecției **tClienti.Rows**, în care căutăm:

```
DataRow linSelectata = tClienti.Rows.Find(cc);
```

unde `cc` este o variabilă `int`, conținând valoarea cheii de căutare pentru `CodClient`.

Explorarea unei relații se bazează pe existența metodelor **GetChildRows și GetParentRows** în clasa **DataRow**, prin care fiecare tuplu (linie de date) vede care linii de date îl leagă și care sunt liniile de date pe care el le leagă, la rândul lui.

Presupunând că avem deja selectată mai sus linia de date despre un client, se pot extrage toți « fiii » acesteia, în cazul nostru comenzile clientului, existente în cadrul relației `Client_Comenzi`:

```
// Navigare spre comenzi folosind relatia Client_Comenzi
DataRow[] vectLinii =
          linSelectata.GetChildRows("Client_Comenzi");
// Afiseaza Clientul
afis.Text+=(cc+" "+linSelectata["NumeClient"]+"      " );
string linia="";
for(int i = 0; i < vectLinii.Length; i++)
      {
      linia="";
      foreach(DataColumn colCrt in ds.Tables["Comenzi"].Columns)
      linia+=vectLinii[i][colCrt]+"    ";
      listBox1.Items.Add(linia);
      cant = (int)vectLinii[i]["Cantitate"];
      Total += cant;
      }
linia=""; linia+="-----------------------";
listBox1.Items.Add(linia);
linia=""; linia+=Total; listBox1.Items.Add(linia);
```

Exercițiu

Să se definească două relații pe o bază de date despre produse, astfel încât să permită afișarea ***Costurilor materiale pe produs*** sub forma unui raport, de genul celui prezentat în figura de mai jos:


```
Form1
[Incarca tabele]   [Stabileste relatii]   [Explorare si calcul]

Costurile materiale pe produs
===============================
p1
------------------------------
      mat_101  1.1   x   101   = 111.1
      mat103   1.3   x   103   = 133.9
      mat104   1.4   x   104   = 145.6
------------------------------
Total pe produs                  390.6
prod_2
------------------------------
      mat 102  2.2   x   102   = 224.4
      mat_101  2.1   x   101   = 212.1
      mat104   2.4   x   104   = 249.6
------------------------------
Total pe produs                  686.1
prod3
------------------------------
      mat 102  3.2   x   102   = 326.4
      mat_101  3.3   x   101   = 333.3
------------------------------
Total pe produs                  659.7
```

Rezolvare

Față de lucrul cu obiecte `Relation` din exemplul de mai sus, problema se complică prin :

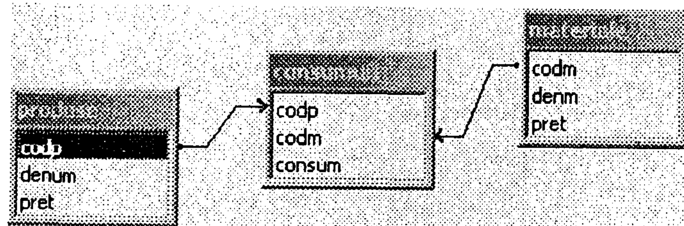
- **preluarea datelor dintr-o bază de date**, nu direct din dataSet;
- existența la nivelul formei a **câte unui adaptor pentru fiecare dintre cele trei tabele** de date *produse*, *materiale*, *consumuri* și a unui singur dataSet :

```
public OleDbDataAdapter daProd, daMat, daCons;
public DataSet dsProdConsMat;
string strConex;
```

instanțiate corespunzător în constructorul formei :

```
strConex=
 "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\\prod.mdb";
daProd=new OleDbDataAdapter("SELECT * from produse",strConex);
daCons=new OleDbDataAdapter("SELECT * from consumuri",strConex);
daMat =new OleDbDataAdapter("SELECT * from materiale",strConex);
dsProdConsMat = new DataSet();
```

- **navigarea folosind două relaţii**, una de la *produse* către *consumuri*, mediată de *cod*, pentru identificarea tuturor consumurilor materiale aferente unui produs şi alta de la *materiale* la *consumuri*, pentru identificarea tuturor consumurilor dintr-un anumit material.



- încărcarea tabelelor în acelaşi dataSet, prin **invocarea fiecărui `dataAdapter`** asociat:

```
private void Incarca_Click(object sender, System.EventArgs e)
{
      daProd.Fill(dsProdConsMat,"produse");
      daMat.Fill(dsProdConsMat,"materiale");
      daCons.Fill(dsProdConsMat,"consumuri");
}
```

Definirea celor două relaţii se poate face cu o funcţie de genul următor:

```
private void Relatii_Click(object sender, System.EventArgs e)
{
 try
  {
   DataRelation drProdToCons = new DataRelation("Prod_Cons",
         dsProdConsMat.Tables["produse"].Columns["codp"] ,
         dsProdConsMat.Tables["consumuri"].Columns["codp"] );
   dsProdConsMat.Relations.Add(drProdToCons);
   textBox1.Text="Relatie Prod_Cons stabilita";
  }
 catch(Exception excpt)
  { textBox1.Text="Esuare relatie ProdToCons "+excpt.Message;}

 try
  {
   DataRelation drMatToCons = new DataRelation("Mat_Cons",
      dsProdConsMat.Tables["materiale"].Columns["codm"] ,
      dsProdConsMat.Tables["consumuri"].Columns["codm"]);
   dsProdConsMat.Relations.Add(drMatToCons);
   textBox1.Text+="\r\nRelatie Mat_Cons stabilita";
  }
```



```
 catch(Exception excpt)
  { textBox1.Text="Esuare relatie MatToCons "+ excpt.Message;}
}
```

în care sunt captate şi semnalate prin mesaje adecvate, eventualele excepţii legate de:

- **nepotrivirea numelui şi a tipului de date** din coloanele care mediază relaţiile;
- existenţa unor **consumuri** ale căror coduri (de produs sau de material) **nu au corespondent în tabelele părinte** etc.

Explorarea relaţiilor, pentru a răspunde cerinţelor informaţionale ale raportului menţionat, se face cu funcţia:

```
private void Calcul_Click(object sender, System.EventArgs e)
{
 textBox1.Text= " Costurile materiale pe produs \r\n"+
      "================================\r\n";
 foreach(DataRow prodCrt in dsProdConsMat.Tables["produse"].Rows)
  {
      decimal valTot=0;
      textBox1.Text += prodCrt["denum"]+
          "\r\n--------------------------------------\r\n";
      DataRow[] vectLinii = prodCrt.GetChildRows("Prod_Cons");
      foreach(DataRow consCrt in vectLinii)
      {
        DataRow linMat= consCrt.GetParentRow("Mat_Cons");
        double valConsum=(double)consCrt["consum"],
                    pretMat=(double)linMat["pret"];
        textBox1.Text+="\t"+ linMat["denm"]+ "\t"+ valConsum+
          "\t x \t"+pretMat+"\t = "+valConsum*pretMat+"\r\n";
        valTot += (decimal)(valConsum*pretMat);
      }
   textBox1.Text +=
          "\r\n--------------------------------------\r\n";
   textBox1.Text +=
          "Total pe produs \t\t\t\t    "+ valTot+"\r\n\r\n";
  }
}
```

Se observă că explorarea relaţiei **`"Prod_Cons"`** s-a făcut pe sensul "**înainte**", identificând toate înregistrările copil (apelul **`prodCrt.GetChildRows("Prod_Cons")`**), în timp ce explorarea relaţiei **`"Mat_Cons"`** s-a făcut pe direcţia "**înapoi**", identificând înregistrarea părinte (apelul **`consCrt.GetParentRow("Mat_Cons")`**).

7. DataRow – tuplu de date

În multe din exemplele anterioare am utilizat un alt tip de obiect necesar lucrului cu baze de date şi anume obiectul **DataRow**. Ne vom opri acum mai mult asupra acestuia. Exemplul de mai jos defineşte un obiect **DataRow** îl configurează conform structurii tabelei **Facturi**, îl încarcă cu date conforme cu tipul fiecărui câmp şi adaugă noua linie la datele tabelei respective:

```
DataRow linNoua ;
linNoua = m_ds.Tables["Facturi"].NewRow();
linNoua["codFactura"] = 101;
linNoua["numeClient"] = "Petrescu I";
linNoua["Data"] = new DateTime(2005, 5, 1);;
m_ds.Tables["Facturi"].Rows.Add(linNoua);
```

Secvenţa de mai jos operează tot cu obiecte **DataRow**, dar le identifică nu prin referinţă ca până acum, ci prin indexare:

```
ds.Tables[1].Rows[3][2].ToString();
ds.Tables["Comenzi"].Rows[3]["Cantitate"].ToString();
```

Se observă că linia de date nu poate fi localizată decât prin poziţie în colecţie, spre deosebire de tabele şi coloane, care pot fi individualizate prin poziţie în colecţie, dar şi prin nume.

Un nume de coloană nu trebuie să conţină caractere speciale (spaţiu, virgulă, punct, ghilimele etc.) şi poate fi specificat în formă completă `SchemaName.OwnerName.TableName`.

Pentru a şterge linii de date dintr-un dataSet trebuie mai întâi, liniile să fie marcate ca şterse, instanţă cu instanţă, invocând metoda **Delete a dataSet-ului.** Ulterior, la apelul **Update pe dataAdapter**, liniile marcate ca şterse vor fi şterse şi din baza de date.

> Metoda **Remove** şterge fizic liniile din colecţia `Rows`; nemaiexistând în colecţie, la momentul apelului `Update` pe dataAdapter, nu vor ma fi găsite ca marcate pentru ştergere, astfel încât ele nu vor fi şterse şi din baza de date.

Editarea unei linii de date: BeginEdit, EndEdit, CancelEdit

Când se efectuează editări masive pe un dataSet, se câştigă timp amânând verificările de restricţii şi declanşarea unor evenimente asociate, până la terminarea tuturor editărilor. Realizăm acest lucru anunţând printr-un apel `BeginEdit()` al liniei. La terminarea editărilor se apelează una din metodele `EndEdit()` sau `CancelEdit()`, în raport cu ce se doreşte: recunoaşterea editărilor sau renunţarea la ele.

Acest lucru explică de ce încălcarea unor restricţii nu este semnalată în blocul `try - catch` propriu operaţiei, ci abia după încheierea editării, la apelul `EndEdit()`.

Proprietatea RowState a unui obiect DataRow

In completarea informaţiei privind valorile **Current** şi **Proposed** ale unei coloane, `DataRow` ţine şi o proprietate **RowState** , care indică starea fiecărei linii în parte: **Added, Deleted, Detached, Modified** sau **Unchanged.**

- **Detached** - o linie este în starea `Detached` cînd a fost creată, dar n-a fost încă adăugată la vreo colecţie `Rows` a vreunui dataSet, sau a fost adăugată, dar apoi a fost scoasă cu `Remove`.
- **Added** – tuplu a fost adăugat la `DataRowCollection`, dar n-a fost încă cerut un `AcceptChanges`.
- **Deleted** – linia a fost stearsă folosind metoda `Delete` a obiectului `DataRow`. Tentativa de a o accesa produce o excepţie **DeletedRowInaccessibleException.**
- **Modified** – una sau mai multe coloane au suferit modificări ale valorilor conţinute, dar n-a fost încă cerut un `AcceptChanges`
- **Unchanged** – linia n-a suferit nici o modificare de la ultimul `AcceptChanges`, terminat cu succes.

Versiunea unui obiect DataRow

Înainte de acceptarea unor schimbări, dataSet-ul ţine ambele versiuni ale liniei de date. Proprietatea **Item** a unei linii de date poate specifica o valoare din enumerarea **DataRowVersion**, pentru a preciza care dată este

cea dorită. Câmpul `Version` poate lua valorile : **`Original`**, **`Default`**, **`Current`** şi **`Proposed`**, cu semnificaţia :

- `Current` – valoarea curentă din linie, modificată sau nu ;
- `Default` – valoarea returnată adecvată stării din proprietatea `RowState`;
- `Original` – valoarea cu care linia a fost încărcată iniţial;
- `Proposed` – valoare modificată a liniei, între momentul cererii `BeginEdit` şi înainte ca `EndEdit` sau `CancelEdit` să fie dat.

Pe durata editării, doar `Current` şi `Proposed` sunt disponibile. După `CancelEdit`, valoarea `Proposed` nu mai este disponibilă. După `EndEdit`, valoarea `Proposed` ia locul celei din `Current`, iar valoarea `Proposed` nu mai este disponibilă.

`AcceptChanges` şi **`RejectChanges`**

Prin apelul `EndEdit()` pe un DataRow nu realizăm efectiv şi schimbările definitive pe linia de date. Consacrarea definitivă a modificărilor se realizează doar printr-un apel `AcceptChanges` sau `RejectChanges`, metode disponibile atât la nivel de DataSet, cât şi la DataTable şi DataRow. Acest tip de apel termină editarea liniilor de date din domeniul respectiv, dând implicit şi `EndEdit()` sau `CancelEdit()`, dacă încă nu se dăduse explicit un astfel de apel.

După apelul **`AcceptChanges()`**, valoarea `Current` se înregistrează ca valoare `Original`, pentru fiecare câmp. Dacă nu s-a apelat încă `EndEdit()`, valoarea `Proposed` este luată în acelaşi timp drept valoare `Current` şi valoare `Original`. Dacă `RowState` indica una din stările `Added`, `Modified` sau `Deleted`, ea va deveni `Unchanged` şi toate modificările devin efective.

După un apel **`RejectChanges()`**, valoarea `Proposed` este ştearsă, renunţându-se definitiv la ea. Dacă `RowState` era una din stările `Deleted` sau `Modified` (adică pentru liniile şterse sau modificate), valorile sunt înlocuite cu cele existente anterior modificărilor, iar `RowState` este pusă pe `Unchanged`. Dacă `RowState` era `Added`, linia este eliminată din colecţia `Rows` a dataSet-ului.

Deoarece după un apel `AcceptChanges()`, `RowState` devine `Unchanged`, apelând în continuare metoda `Update` a DataAdapter-ului, nu se va mai opera nici-o modificare în baza de date. În consecinţă, apelul metodei **`Update` a DataAdapter-ului trebuie să preceadă totdeauna apelul `AcceptChanges`** pe oricare DataRow, DataTable sau DataSet.

Uzual, **`AcceptChanges` pe DataSet se apelează doar dacă metoda `dataAdapter1.Update` returnează un cod de succes** ; în caz contrar este captată o excepţie, în interiorul căreia se apelează `RejectChanges`.

Dacă nu rejectăm schimbările în cazul eşecului operării lor în baza de date, liniile rămân modificate în DataSet, astfel că la următorul apel de `Update`, cererea va fi din nou refuzată, deoarece pe lângă liniile nou modificate, cele vechi neoperate sunt încă tot în aşteptarea actualizării. Deoarece DataSet-ul este gândit să lucreze şi independent de baza de date, **faptul că un `Update` este operat cu succes pe baza de date, nu are nimic a face cu acceptarea sau refuzul schimbărilor în liniile de date ale DataSet-ului.**

`Transactions` şi **`Updates`**

Operaţiile cerute printr-un `Update` nu se execută neapărat într-o singură tranzacţie, lucru important când la baza de date sunt conectaţi simultan mai mulţi utilizatori. Pentru a forţa efectuarea întregului `Update` într-o singură tranzacţie **conexiunea** trebuie să anunţe `BeginTransaction`, moment în care primeşte un obiect **`SqlTransaction`**. Acest obiect dispune la rândul lui de metodele `Commit` şi `Rollback` pentru închiderea tranzacţiei.

Succesiunea de derulare va fi în acest caz: `Open` pe conexiune, `BeginTransaction` pe conexiune, `Commit` sau `Rollback` pe obiectul **Transaction**; `Close` conexiune.

8. DataView – vizualizarea

Clasa **`DataView`** oferă un mod de vizualizare a unui obiect dataSet. Ca funcţionalitate, această clasă oferă posibilitatea filtrării / sortării datelor dintr-o tabelă; se comportă ca o colecţie de linii şi are conţinutul bazat pe o condiţie dintr-o frază **Select**, cu avantajul că deţinând obiecte de sine

stătătoare, permite configurarea atât static (în design), cât şi la momentul execuţiei.

Mecanismul este simplu: se instanţiază un obiect DataView pornind de la o tabelă din dataSet şi se fixează proprietăţile dataView-ului, printre care şi `RowFilter` sau `RowStateFilter`, când selecţia se bazează pe starea unei linii de date (în curs de modificare, ştearsă, nou adăugată etc.)

Aceleaşi tabele ale unui DataSet pot avea mai multe vizualizări asociate, reprezentând puncte de vedere diferite ale unor utilizatori şi în raport cu acestea se vor folosi diverse criterii de selecţie.

În plus, un obiect dataView poate fi legat de controale cu vizualizare, cum ar fi `ComboBox`, `ListBox`, `DataGrid` etc., asigurând descrierea facilă a părţii de interactivitate a aplicaţiilor cu baze de date.

Exerciţiu

Folosind conceptul de DataView să se afişeze într-un textBox produsele cu un preţ peste o limită dată, produsele nou adăugate într-o sesiune de lucru, respectiv toate produsele dintr-o bază de date.

Rezolvare.

1. Într-o aplicaţie C# Windows App se pun:
   - `textBox1` cu proprietăţile `Multiline = true`, `Dock = Right`, `ScrollBars = Both`, `ReadOnly = true`;
   - `button1` inscripţionat „`Filtreaza`"

2. În clasa `Form1` se declară un `DataSet` şi un `OleDbDataAdaptor`:

```
OleDbDataAdapter ProduseAD;
DataSet ProduseDS;
```

care se instanţiză în constructorul clasei `Form1`, folosind apoi la încărcarea datelor despre produse:

```
ProduseDS = new DataSet("setul_meu");
string frazaSQL = "select codp, denum, pret from produse";
string sirConex =
 "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\\prod.mdb";
ProduseAD= new OleDbDataAdapter(frazaSQL, sirConex);
ProduseAD.Fill(ProduseDS,"produse");
```



```
Form1
[Filtreaza]   [Sorteaza]

Lista produselor scumpe
===========================
5    pr_5    500
6    pr_6    600
Lista produselor adaugate
===========================
x1   Prod No1 400
x2   Prod No2 300
Lista produselor scumpe
===========================
5    pr_5    500
6    pr_6    600
x1   Prod No1 400
Lista produselor sortate
===========================
1    pr_1    100
2    pr_2    200
x2   Prod No2 300
3    pr_3    300
x1   Prod No1 400
5    pr_5    500
6    pr_6    600
```

Pentru tratarea evenimentului Click pe butonul `Filtreaza` se scrie funcţia:

```
private void Filtreaza_Click (object sender, System.EventArgs e)
{
    DataView Scumpe =
            new DataView( ProduseDS.Tables["Produse"] );
    Scumpe.RowFilter = "(pret > 300) AND (pret < 1500)";

    DataView ProduseAdaugate =
             new DataView( ProduseDS.Tables["Produse"] );
    ProduseAdaugate.RowStateFilter=DataViewRowState.Added;

    PrintProduse( Scumpe, "Lista produselor scumpe" );

    DataRow linNoua = ProduseDS.Tables["Produse"].NewRow();
    linNoua["codp"] = "x1";  linNoua["denum"] = "Prod No1";
    linNoua["pret"] = 400;
```

```
        ProduseDS.Tables["Produse"].Rows.Add( linNoua );

        linNoua = ProduseDS.Tables["Produse"].linNoua();
        linNoua["codp"] = "x2";  linNoua["denum"] = "Prod No2";
        linNoua["pret"] = 300;
        ProduseDS.Tables["Produse"].Rows.Add( linNoua );

        PrintProduse
              ( ProduseAdaugate, "Lista produselor adaugate");
        PrintProduse( Scumpe, "Lista produselor scumpe");
}
```

Pentru facilitarea afişării în textBox, se va scrie o funcţie de forma :

```
private void PrintProduse( DataView crtView, string titlu)
{
  textBox1.Text += "\r\n\r\n"+ titlu
                      +"\r\n=========================== ";
  textBox1.Text+="\r\n \r\n";
  for (int i=0; i< crtView.Count; i++)
   {
      textBox1.Text+="\r\n\t"+ crtView[i]["codp"]+
             "\t"+crtView[i]["denum"]+"\t"+crtView[i]["pret"];
   }
}
```

Ieşirea afişată pe textBox va arăta ca în figura de mai sus.

S-au folosit proprietăţile `RowFilter` care specifică expresia ce stă la baza filtrării:

```
Scumpe.RowFilter = "(pret > 300) AND (pret < 1500)";
```

respectiv `RowStateFilter`, care selectează înregistrările pe baza stării lor (`Added, Deleted, Unchanged` etc.):

```
ProduseAdaugate.RowStateFilter = DataViewRowState.Added;
```

Obiectul DataView este "bindable" ceea ce îi conferă o mare utilitate în manipularea unor subseturi de date, vizualizabile în grid. De remarcat că modificările au rămas la nivel de DataSet, nu s-au propagat şi în baza de date, dar acest lucru este posibil printr-un apel `Update()`, specific adaptorului `ProduseAD`.

Fiecărei tabele îi pot fi asociate mai multe DataView-uri; o tabelă dispune de cel puţin o vizualizare implicită, cea înregistrată şi în

proprietatea DefaultDataView. Proprietăţile acesteia pot fi setate doar la momentul execuţiei, nu şi static, cu Designer-ul.

Deşi se aseamănă cu DataRow, liniile unui dataView sunt de tip **DataRowView**, deoarece ele pot avea şi alte proprietăţi, specifice doar vizualizării:

- **DataView** – vizualizarea căreia îi aparţine linia;
- **IsEdit** – true, când linia a fost editată
- **IsNew** - true când `DataRowView` este nou adăugată
- **[index]** – indexarea ce returnează valoarea unei coloane din `DataRowView`
- **Row** - `DataRow` căreia îi asigură vizualizarea
- **RowVersion** - versiunea curentă a `DataRowView`

```
DataRowView dvr = ToateSortate[2]; // selectarea linie 2
MessageBox.Show(dvr[2].ToString());// afisarea coloanei 2
```

Sortarea înregistrărilor unui DataView

Sortarea înregistrărilor unui DataView se face simplu, indicând în proprietatea `Sort` a vizualizării, expresia de sortare, ca o listă ordonată de coloane, eventual urmate de cuvintele cheie **ASC** sau **DESC**, indicând ordinea de sortare :

```
private void btnSorteaza_Click(object sender, EventArgs e)
{
  DataView ToateSortate =
               new DataView( ProduseDS.Tables["Produse"] );
  ToateSortate.Sort="pret, denum DESC";
  PrintProduse(ToateSortate , "Lista produselor sortate");
}
```

- Select întoarce un array DataRows

DataViewManager lucrează similar DataSet -ului, fiind un container pentru DataViews.

9. Obiecte de tip Command

Clasele de tip command (`SqlCommand, OleDbCommand`) sunt folosite pentru a executa o comandă SQL sau proceduri stocate. Constructorii lor primesc o instrucţiune SQL sau numele procedurii stocate şi un obiect

conexiune; există versiuni de supraîncărcare ce primesc doar unii dintre parametrii de intrare, ceilalți putând fi adăugați ulterior.

```
OleDbCommand cmd =
        new OleDbCommand("SELECT * FROM produse", con);
```

sau

```
SqlCommand cmd = new SqlCommand();
cmd.CommandText = "SELECT * FROM Clienti";
cmd.Connection = con;
```

Obiectele de tip Command reprezintă unica modalitate prin care în ADO.NET se execută comenzi pe o sursă de date; sunt utile execuției comenzilor de INSERT, UPDATE și DELETE, fiind înregistrate ca proprietăți la nivelul obiectelor DataAdapter; ele pot genera însă și obiecte **DataReader** și **XMLDataReader** (de tipuri adecvate provider-ilor: `OleDbDataReader`, `SqlDataReader` etc.), putând accesa baza de date fără a fi nevoie de un DataAdapter. În acest sens, obiectele XxxCommand preiau ele prin constructori, în locul adaptorului, string-ul de conectare și conexiunea:

```
OleDbCommand myCmd = new OleDbCommand(strSql,con);
```

În plus, obiectele de tip Command dețin o colecție `Parameters`, ce reunește parametrii necesari unor comenzi mai sofisticate și care sunt necesari și apelului de proceduri stocate.

Clasele de tip XxxParameter (`SqlParameter, OleDbParameter`) sunt cele care permit adăugarea de parametri la o comandă; parametrii au un nume, un tip, o direcție input sau output și o valoarea; ei se adaugă la colecția `ParametersCollection` `Parameters` pe care o deține fiecare comandă.

Direcția este implicit input, dar poate fi dată și explicit.

Detalii privind clasele de tip xxxParameter sunt date mai jos, în paragrafele privind utilizarea procedurilor stocate.

Comenzile de acces direct la baza de date sunt de trei tipuri :

- `ExecuteReader`, care întoarce o colecție de linii de date, accesibile apoi linie cu linie;
- `ExecuteScalar`, care returnează o singură valoare, de tip generic `object`;

- `ExecuteNonQuery`, care nu returnează nimic, dar execută actualizările asupra bazei de date, generate prin modificări, ștergeri sau inserări în baza de date.

Obiectele claselor de tip Reader (`SqlDataReader`, `OleDbDataReader`) pot fi generate de către obiectele de tip Command prin apelul `ExecuteReader` și furnizează un flux de date, read-only, disponibilizând câte o înregistrare la fiecare apel al metodei `Read()`, spre deosebire de metoda Fill() a unui adaptor, care furnizează o colecție de înregistrări :

```
OleDbDataReader dr = myCmd.ExecuteReader();
```

`DataReader`-ul pointează aprioric pe BOF și e nevoie de o citire prealabilă pentru a avea o înregistrare.

Parcurgerea cu DataReader este simplă și se aseamănă cu citirea câte unei înregistrări dintr-un fișier. Localizarea unui câmp în înregistrarea din DataReader se face prin indexare, sau cunoscând numele coloanei din tabelă.

```
textBox1.Text = dr["pret"];
```

Funcția de mai jos, pusă pe butonul unei forme, listează într-un TextBox multilinie, docat dreapta, un catalog al produselor dintr-o bază de date :

```
private void btnCatalog_Click
                    (object sender, System.EventArgs e)
{
 string strSql = "SELECT codp, denum, pret FROM produse";
 OleDbConnection con =
 new OleDbConnection(
   @"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\prod.mdb");
 try {con.Open(); }
 catch { MessageBox.Show("Eroare open conexiune");}

 OleDbCommand myCmd    =new OleDbCommand(strSql,con);

 OleDbDataReader dr = myCmd.ExecuteReader();
 textBox1.Text="";
 while (dr.Read())
     {
      textBox1.Text+="\r\n"+dr["codp"]+
           " -- " + dr["denum"]+ " costa  " + dr["pret"];
     }
 dr.Close();
}
```

Se observă că nu se folosesc obiecte DataAdapter sau DataSet, ci doar obiecte XxxCommand şi DataReader.

Pentru exemplificarea unei comenzi `ExecuteScalar()` am presupus că dorim să afişăm numărul de produse distincte, aflate în tabela produse. În acest scop am pus pe un buton din macheta aplicaţiei, următoarea funcţie :

```
private void btnExecScalar_Click
                        (object sender, System.EventArgs e)
{
 OleDbConnection conn = null;
 OleDbCommand myCommand = null;
 try
 {
  conn = new OleDbConnection(
  @"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\prod.mdb");
  conn.Open();
  myCommand = new OleDbCommand
              ( "SELECT COUNT(*) FROM produse", conn);
  int nrProd = (int)myCommand.ExecuteScalar();
  textBox1.Text="Numar produse comercializate: "+ nrProd;
 }
 catch(Exception excpt)
 {
      MessageBox.Show
              ("Esec pe ExecuteScalar: "+excpt.Message);
 }
}
```

Un apel `ExecuteNonQuery()` a fost exemplificat presupunând că se doreşte modificarea denumirii produselor care încep cu « p », prin adăugarea prefixului « txt ». Funcţia care realizează acest lucru tratează evenimentul `Click` al unui buton inscripţionat sugestiv:

```
private void btnExecNonQuery_Click
                        (object sender, System.EventArgs e)
{
      OleDbConnection conn = null;
      OleDbCommand command = null;

      string ConnString =
            "Provider=Microsoft.Jet.OLEDB.4.0;"+
                              "Data Source=C:\\prod.mdb";
      string cmd =
            "UPDATE produse SET denum = 'txt'&denum "+
                                  "where denum like 'p%'";
      try
      {
```



```
        conn = new OleDbConnection(ConnString);  conn.Open();
        command = new OleDbCommand(cmd, conn);
        int nrRec = command.ExecuteNonQuery();
        textBox1.Text="Numar inregistrari afectate: "+ nrRec;
      }
      catch(Exception excpt)
      {
            MessageBox.Show
               ("Esec pe ExecuteNonQuery: "+ excpt.Message);
      }
      finally
      {
        if (conn.State == ConnectionState.Open) conn.Close();
      }
}
```

10. Lucru cu procedurile stocate

Procedurile stocate sunt cereri frecvent folosite asupra unei baze de date, grupate sub un nume de apel. Procedurile stocate au două avantaje majore:

- sunt stocate pe server şi nu au nevoie să fie compuse şi retransmise de fiecare dată când sunt invocate ;
- sunt **deja compilate**, fiind lansate direct în execuţie şi beneficiind deci şi de utilizarea **statisticilor interne** pentru optimizarea accesului la datele unei tabele.

```
private void creareProc_Click(object sender, System.EventArgs e)
{
 string strCreare = "CREATE PROCEDURE ProdScumpe AS "
   +"Select * From produse where pret > 600 Order by denum";
 OleDbConnection con =
  new OleDbConnection(
   @"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\prod.mdb");
 con.Open();

 OleDbCommand myCmd = new OleDbCommand();
 myCmd.CommandType = CommandType.Text;
 myCmd.Connection = con;
 myCmd.CommandText="DROP PROCEDURE ProdScumpe";

 OleDbDataReader dr;
 try { dr = myCmd.ExecuteReader(); dr.Close(); }
 catch { MessageBox.Show("Procedura de sters nu exista!!");}

 myCmd.CommandText = strCreare;
```

```
 try { dr = myCmd.ExecuteReader();dr.Close();}
 catch { MessageBox.Show("Procedura nu a fost creata!!");}
}
```

Se pot folosi diverse modalități de a apela o procedură stocată; cel mai simplu mod este să înlocuim instrucțiunea SELECT cu numele procedurii stocate:

```
private void apelProc_Click(object sender, System.EventArgs e)
{
 OleDbConnection con =
   new OleDbConnection(
   @"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\prod.mdb");
 con.Open();

 OleDbCommand myCmd = new OleDbCommand("ProdScumpe",con);
 myCmd.CommandType = CommandType.StoredProcedure;
 OleDbDataReader dr = myCmd.ExecuteReader();
 textBox1.Text="Lista produselor scumpe";
 while (dr.Read())
     {
      textBox1.Text+="\r\n"+dr["codp"]+ " -- " +
                 dr["denum"]+ " costa " +dr["pret"];
     }
 dr.Close();
}
```

Dacă procedura ar lucra cu parametri de intrare, spre exemplu numai produsele cu pret peste **o valoare, dată ca text într-un TextBox al formei**, procedura s-ar schimba deoarece comanda trebuie să declare acum și parametri de intrare / ieșire; în acest caz este nevoie să lucrăm și cu obiecte de tip `Parameter`.

Așa cum spuneam, obiectele Command dispun de o colecție numită **`Parameters`**, în care prin metoda **`Add`**, adăugăm parametrii pregătiți anterior sau furnizăm metodei informația necesară pentru a construi ea acești parametri:

```
OleDbParameter param1, param2;

param2 = new OleDbParameter("@NrProd", OleDbType.Integer);
param2.Direction = ParameterDirection.Output;
cmdMea.Parameters.Add(param2);

param1 = cmdMea.Parameters.Add(
      new OleDbParameter("@PretMin", OleDbType.Integer) );
```


Metoda **`Add`** are mai multe versiuni supraîncărcate, în funcție de numărul de argumente folosite la construirea unui obiect parametru.

Se observă că în afara faptului că metoda a construit și a adăugat un parametru de tip `int` la o comandă, ea returnează și referința parametrului, astfel încât acesta să poată fi ulterior echipat și cu alte proprietăți, spre exemplu direcția:

```
param1.Direction = ParameterDirection.Input;
```

Parametrul mai este accesibil și prin metoda de **indexare** a colecției, care localizează un parametru după **poziție** sau după **numele** său; construcția de mai jos stochează drept valoare pentru parametru de intrare în procedură, prețul rezultat prin conversia în `int` a textului introdus de utilizator la rulare, în câmpul **`txtPret`**:

```
      cmdMea.Parameters["@PretMin"].Value =
                Convert.ToInt32(txtPret.Text);
```

sau

```
      cmdMea.Parameters[0].Value =
                Convert.ToInt32(txtPret.Text);
```

În ADO.NET parametri trebuie **să se numească la fel** cu cei definiți în procedura stocată, să aibă **același tip** și **aceeași lungime**.

Direcția poate fi:

- **`Input`** – parametru de intrare în procedură;
- **`Output`** - parametru de ieșire, returnat de procedură și care nu poate conține date de intrare pentru procedură;
- **`InputOutput`** - parametru folosit în același timp și pentru intrare și pentru ieșire, în comunicarea cu procedura;
- **`ReturnValue`** - parametru de ieșire, dar limitat la unul singur.

Pot exista mai mulți parametri de tip **`Input, Output, InputOutput`**, dar doar un singur parametru de tip **`ReturnValue.`**

Funcțiile de creare, respectiv folosire a unei **proceduri stocate cu parametru de intrare**, arată acum astfel:

```
private void creareProc_Click
                    (object sender, System.EventArgs e)
```

```
{
 string strCreare =
      "CREATE PROCEDURE ProdPestePret(@pretMin INT) AS " +
           "SELECT * FROM produse WHERE pret > @pretMin " ;

 OleDbConnection con =
      new OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;"
                               +@Data Source=C:\prod.mdb" );
 con.Open();

 OleDbCommand cmdMea    =new OleDbCommand();
 cmdMea.CommandType = CommandType.Text;
 cmdMea.Connection = con;
 OleDbDataReader dr;

      // daca exista, o stergem pentru a crea o noua versiune
 cmdMea.CommandText="DROP PROCEDURE ProdPestePret";
 try { dr = cmdMea.ExecuteReader(); dr.Close();}
 catch(Exception excpt)
  {
       MessageBox.Show("Stergere esuata: "+ excpt.Message );
  }

      // refolosire comanda pentru a crea noua procedura
 cmdMea.CommandText = strCreare;
 cmdMea.CommandType = CommandType.Text;

 try { dr = cmdMea.ExecuteReader();dr.Close() ;}
 catch(Exception excpt)
  {
      MessageBox.Show("Creare esuata: "+excpt.Message);
  }
}

private void apelProc_Click(object sender, System.EventArgs e)
{
 OleDbConnection con = new OleDbConnection(
  @"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\prod.mdb"
                                           );
 con.Open();

 OleDbCommand cmdMea    =
           new OleDbCommand("ProdPestePret "+txtPret, con);
 cmdMea.CommandType=CommandType.StoredProcedure;

 OleDbParameter param1;
 param1 = cmdMea.Parameters.Add(
           new OleDbParameter("@PretMin", OleDbType.Integer));
 param1.Direction = ParameterDirection.Input;
```



```
 cmdMea.Parameters["@PretMin"].Value =
                          Convert.ToInt32(txtPret.Text);

 OleDbDataReader dr = null;

 try {  dr = cmdMea.ExecuteReader(); }
 catch(Exception excpt)
      {
           MessageBox.Show(excpt.Message);
      }
 textBox1.Text="Lista produselor peste "+ txtPret.Text +
                 " EUR \r\n\r\n";
 while (dr.Read())
      {
           textBox1.Text+="\r\n"+dr["codp"]+ "   " +
                 dr["denum"] +"     " + dr["pret"]+ " EUR";
      }
 dr.Close();
}
```

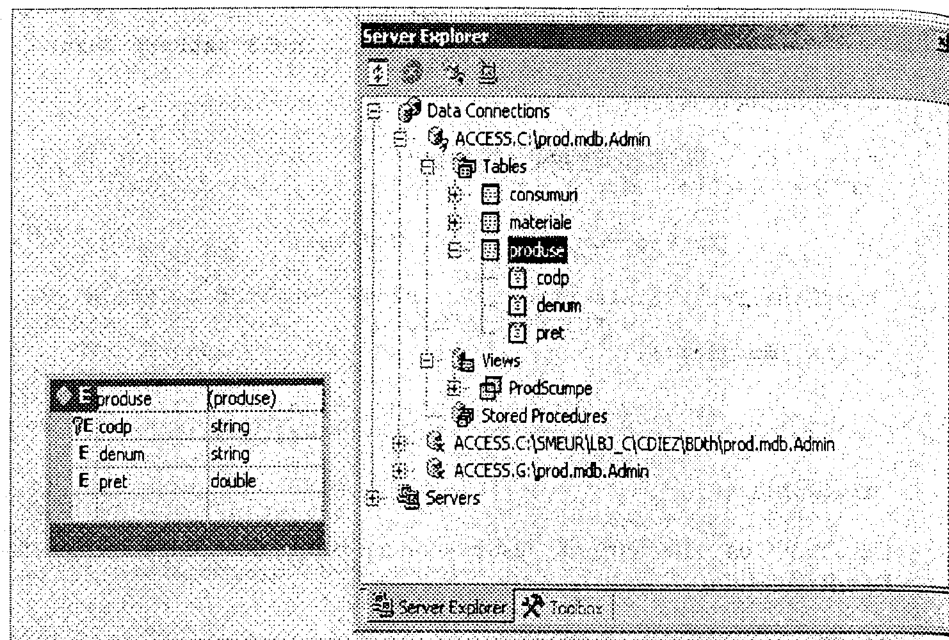
De reținut că parametrii de ieşire sunt accesibili doar după închiderea DataReader-ului.

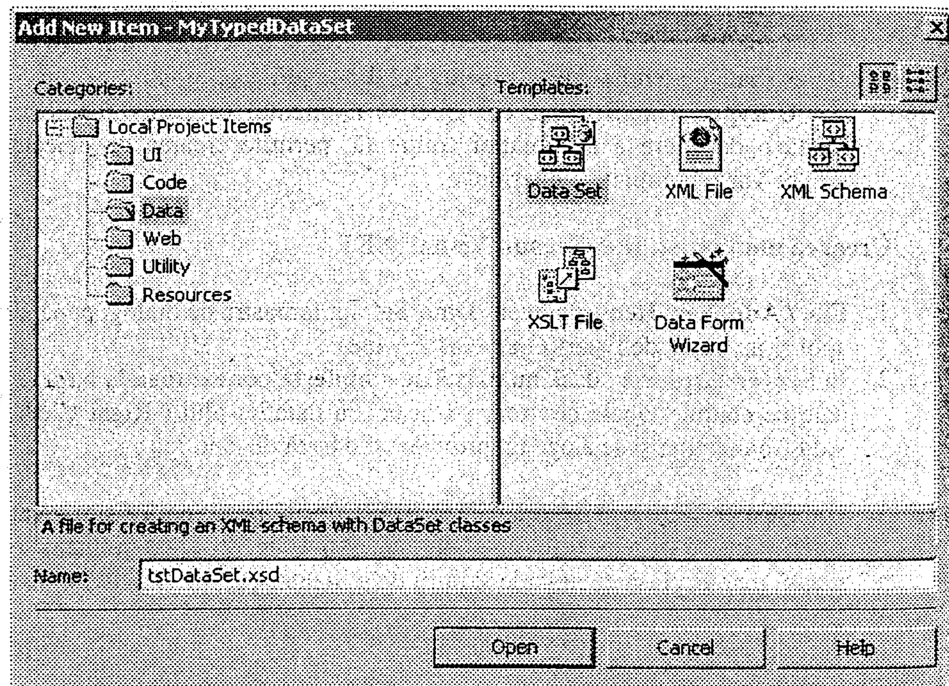
11. Typed şi untyped DataSet

- **Typed DataSet** – un DataSet care conține şi descrierea structurii de date asociată (eventual într-o schemă XML ataşată).
- **Untyped DataSet** este cel creat la momentul execuției şi nu are ataşată o descriere a informației conținută; permite crearea de obiecte vide, ce vor fi completate ulterior.

Crearea unui typed DataSet sub Visual .NET

1. **File / Add New Item** şi alegem **Data Set** din fereastra de dialog, punând apoi şi un nume de fişier (`tstDataSet.xsd`).
2. În **Server Explorer**, dacă nu există **se stabileşte conexiunea** la baza de date ce conține tabela ce vrem s-o asociem dataSet-ului (Right Click, Add Connection şi se alege un provider şi o bază de date) ;
3. Din **Server Explorer** se draghează tabela dorită peste noul fişier creat anterior cu Add Item şi deschis automat în mediu. În acest moment se crează o nouă clasă cu numele stabilit de utilizator (`tstDataSet`).





În mediu va apare forma vizuală a descrierii, iar în fişierul `tstDataSet.xsd` de descriere, găsim schema XML asociată :

```
<xs:element name="produse"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="codp" type="xs:string" />
            <xs:element name="denum" type="xs:string"
minOccurs="0" />
            <xs:element name="pret" type="xs:double"
minOccurs="0" />
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

De altfel din **bara de stare a ferestrei de afişare a fişierului** se poate alege forma de vizualizare (`DataSet` sau `XML`).

Lucru cu typed DataSet – ul creat mai sus este extrem de facil acum; este de ajuns să instanţiem un obiect :

```
tstDataSet ds = new tstDataSet();
```

şi vedem că el conţine ca elemente toate tabelele dragate peste acel DataSet, iar pentru fiecare tabelă au fost construite metode (`AddproduseRow`, `NewproduseRow` etc.), au fost disponibilizate evenimente (`produseRowChanged`, `produseRowDeleted` etc.), iar structura permite acces rapid la orice câmp din bază, printr-un nume pus automat de către sistem (`pretColumn`):

```
ds.produse.AddproduseRow("120","prod120",120.12);
textBox1.Text = ds.produse.pretColumn.ToString()+" = ";
textBox1.Text+=
  ds.produse.Rows[0][ds.produse.pretColumn].ToString();
```

Chiar lucru cu un DataAdapter este acum simplificat; este suficient să tragem din ToolBox un DataAdapter şi să răspundem întrebărilor puse de wizard şi vom avea la final un adaptor de date care poate fi încărcat printr-o simplă comandă de `Fill`.

```
oleDbDataAdapter1.Fill(ds);
```

Pentru testare vom afişa denumirea tuturor produselor (produsul existent deja în dataSet, din adăugarea anterioară, plus cele adăugate prin apelul **Fill**) :

```
foreach(DataRow lin in ds.produse.Rows)
      textBox1.Text+=
            "\r\n"+lin[ds.produse.denumColumn].ToString();
```

comparată cu secvenţa următoare unde `tstDataSet.produseRow` este văzut ca tip distinct, care are în compunere chiar coloanele tabelei:

```
foreach(tstDataSet.produseRow lin in ds.produse.Rows)
            textBox1.Text+="\r\n"+ lin.pret;
```


APLICAŢII CU BAZE DE DATE SUB ADO.NET

1. **Legarea câmpurilor dintr-o tabelă a bazei de date cu proprietăţile unor controale din macheta unei aplicaţii**
2. **Abordarea programatică a lucrului cu baze de date sub ADO.NET**
3. **Abordarea vizuală a lucrului cu baze de date**

1. Legarea câmpurilor dintr-o tabelă a bazei de date cu proprietăţile unor controale din macheta unei aplicaţii

După cum am văzut deja într-un capitol anterior, există două tipuri de legături ce se pot stabili între nişte surse de date şi diverse proprietăţi ale unor controale:

- **simple data binding** realizată pe controale obişnuite, gen Label, TextBox sau Button; se leagă doar o singură valoare (există doar una cum ar fi cea returnată de **Select Count (*)** sau selectăm una din mai multe, valoare pe care o vom denumi valoare/înregistrare curentă).
- **complex data binding** acceptată de controale bazate pe colecţii: ListBox, ComboBox sau DataGrid, capabile să afişeze mai multe valori simultan.

Ne propunem acum să vedem cum lucrează acest mecanism în cazul bazelor de date, atât în varianta simplă, cât şi complexă.

Legătură simplă realizată vizual

Spre deosebire de masive, sursele de date structurate în baze de date **sunt vizibile şi în Designer**, astfel încât legările simple sau complexe se pot face şi vizual.

1. Intr-o aplicaţie nouă C# Windows Application punem un **textBox1** şi un **button1**.
2. Vedem în Server Explorer o conexiune cu BD despre produse, materiale si consumuri specifice. Dacă nu, se crează şi o conexiune în Server Explorer cu *MouseRight* pe **Data Connections / Add** şi alegem MS Jet 4.0 ca provider, iar conexiunea trimite la fişierul ***prod.mdb.***

3. Tragem peste Form tabela **materiale**, iar pe **oleDataAdapter1** pus automat de mediu, adăugăm proprietatea **SelectCommand** conţinând comanda:

```
SELECT COUNT(*) AS total FROM materiale
```

4. Generam dataset-ul **dataSet11** folosind RightClick pe **oleDataAdapter1** + **Generate DataSet**

5. Proprietatea Text a **textBox1** o legăm dinamic de variabila total obţinută la umplerea setului de date, astfel: pe **Properties** cu textBox1 selectat alegem **Data Binding / Advanced** şi deschidem colecţia în care vedem posibile câmpuri de legat, dar nu şi **total**, **căci acesta este un rezultat al execuţiei.**
6. Activăm setul de date folosind RightClick pe **oleDataAdapter1** / **Generate DataSet.**
7. Pe **Properties** textBox1 alegem **Data Binding / Advanced** şi deschidem colecţia în care de data aceasta vedem şi câmpul **total** pe care îl selectăm ca fiind legătura pentru proprietatea **Text** a controlului **textBox1**.
8. Pe **button1**, eveniment Click, umplem setul de date:

```
oleDbDataAdapter1.Fill(dataSet11);
```

In textBox1 la rulare va apare numărul înregistrărilor citite.

Observaţie.

Prin legătură simplă se poate lega şi un element dintr-o colecţie, chiar dacă nu defilam prin toată colecţia; în acest caz se afişează implicit numai primul element din colecţie, dar vom putea naviga prin colecţie folosind un obiect **CurrencyManager**, aşa cum se va vedea în subcapitolul următor.

Legătură simplă realizată prin program

1. Dacă nu există, se crează baza de date *prod.mdb* cu tabela **produse** şi câmpurile **codp** (text), **denum** (text) şi **pret** (double).
2. Eventual se crează şi o conexiune în Server Explorer cu *MouseRight* pe **Data Connections / Add** şi alegem MS Jet 4.0 ca provider, iar conexiunea trimite la fişierul *prod.mdb*

3. Pe o aplicaţie Windows Forms se pun trei textBox-uri: **tb_cod**, **tb_denum** şi **tb_pret**
4. Punem în antet **using System.Data.OleDb;**
5. Adăugăm două butoane (stânga, dreapta), pentru navigare
6. Se pun în clasa Form1 doi membri referinţă, pentru un set de date şi un manager de legătură între câmpurile din machetă şi datele din setul de date:

```
private CurrencyManager crtManager;
private DataSet dataSetul;
```

7. Pe constructorul lui Form1, creăm un set de date şi două string-uri, unul cu fraza SQL şi celălalt cu parametrii conexiunii la baza de date:

```
dataSetul = new DataSet( );
string frazaSQL =
          "select codp, denum, pret from produse";
string sirConex = "Provider=Microsoft.Jet.OLEDB.4.0;" +
                              "Data Source=C:\\prod.mdb;";
```

Şirul de conectare se poate lua cu Copy / Paste din **Properties** aferente conexiunii (MouseRight în Server Explorer pe conexiune şi aleg proprietatea **ConnectString**).

8. În continuare, tot pe constructorul lui Form1 creăm un adaptor de date şi cerem umplerea setului de date, folosind un cuplu try – catch pentru eventualitatea că baza de date nu se găseşte în directorul căutat:

```
try
{
      OleDbDataAdapter Adaptorul =
            new OleDbDataAdapter(frazaSQL, sirConex);
      Adaptorul.Fill(dataSetul, "tabela");
}
catch
 { MessageBox.Show("BD prod.mdb nu este la locul ei"); }
```

9. Adăugăm în constructorul lui Form1 legatura simplă a proprietăţii Text a celor trei TextBox-uri cu câte un câmp din tabelă :

```
tb_cod.DataBindings.Add
      ("Text",dataSetul.Tables["tabela"],"codp");
```

```
tb_denum.DataBindings.Add
        ("Text",dataSetul.Tables["tabela"],"denum");
tb_pret.DataBindings.Add
        ("Text",dataSetul.Tables["tabela"], "pret");
```

10. Cerem binding - context-ului referința managerului curent de legături ce gestionează legăturile de mai sus, făcute pe sursa de date identificată prin `dataSetul` nostru :

```
crtManager = (CurrencyManager)
        this.BindingContext[dataSetul.Tables["tabela"]];
```

11. Adăugăm două funcții pentru înapoi-înainte, pe lista managerului curent de legături:

```
protected void btn_stg_onclick(object sender, EventArgs e)
    {
        m_lm.Position -= 1; // previous in lista manager
    }

protected void btn_drt_onclick(object sender, EventArgs e)
    {
        m_lm.Position += 1; // next in lista manager
    }
```

12. Adăugăm manual codul, în `InitializeComponent`, pe delegatul corespunzător funcțiilor de tratare a apăsării pe butoane **sau le adăugăm vizual** ca funcții de tratare `Click`, cu **Properties / Events**, corespunzător fiecărui buton:

```
this.btn_stg.Click +=
        new System.EventHandler(this.btn_stg_onclick);
this.btn_drt.Click +=
        new System.EventHandler(this.btn_drt_onclick);
```

Programul sursă arată acum astfel:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Data.OleDb;

namespace BD_Binding
{
```



```
    public class Form1 : System.Windows.Forms.Form
    {
    private CurrencyManager crtManager;
    private DataSet dataSetul;
    private System.Windows.Forms.TextBox tb_cod;
    private System.Windows.Forms.TextBox tb_denum;
    private System.Windows.Forms.TextBox tb_pret;
    private System.Windows.Forms.Button btn_stg;
    private System.Windows.Forms.Button btn_drt;

    private System.ComponentModel.Container components=null;

  public Form1()
   {
    InitializeComponent();
    dataSetul = new DataSet( );
    string frazaSQL = "select codp, denum, pret from produse";
    string sirConex =
       "Provider=Microsoft.Jet.OLEDB.4.0;" +
                        "Data Source=C:\\prod.mdb;";
    try
     {
       OleDbDataAdapter Adaptorul =
             new OleDbDataAdapter(frazaSQL, sirConex);
       Adaptorul.Fill(dataSetul, "tabela");
     }
   catch
      { MessageBox.Show("BD prod.mdb nu este la locul ei"); }

// Leaga proprietatea Text a textBox-urilor de campuri din BD
   tb_cod.DataBindings.Add
            ("Text",dataSetul.Tables["tabela"],"codp"
   tb_denum.DataBindings.Add
            ("Text",dataSetul.Tables["tabela"],"denum");
   tb_pret.DataBindings.Add
            ("Text",dataSetul.Tables["tabela"], "pret");

// cerem  managerul curent al contextului de legare
   crtManager = (CurrencyManager)
             this.BindingContext[dataSetul.Tables["tabela"]];
  }

protected void btn_stg_onclick(object sender, EventArgs e)
  {
      crtManager.Position -= 1; // previous in lista manager
  }
protected void btn_drt_onclick(object sender, EventArgs e)
  {
      crtManager.Position += 1; // next in lista manager
  }
```

```
protected override void Dispose( bool disposing )
 {
  if( disposing )
       {
            if (components != null)
                 {
                      components.Dispose();
                 }
       }
  base.Dispose( disposing );
  }

 #region Windows Form Designer generated code

 private void InitializeComponent()
 {
  this.tb_cod = new System.Windows.Forms.TextBox();
  this.tb_denum = new System.Windows.Forms.TextBox();
  this.tb_pret = new System.Windows.Forms.TextBox();
  this.btn_stg = new System.Windows.Forms.Button();
  this.btn_drt = new System.Windows.Forms.Button();
  this.SuspendLayout();
       //
       // tb_cod
       //
  this.tb_cod.Location = new System.Drawing.Point(56, 40);
  this.tb_cod.Name = "tb_cod";
  this.tb_cod.TabIndex = 0;
  this.tb_cod.Text = "tb_cod";
       //
       // tb_denum
       //
  this.tb_denum.Location = new System.Drawing.Point(168, 40);
  this.tb_denum.Name = "tb_denum";
  this.tb_denum.Size = new System.Drawing.Size(240, 20);
  this.tb_denum.TabIndex = 1;
  this.tb_denum.Text = "tb_denum";
       //
       // tb_pret
       //
  this.tb_pret.Location = new System.Drawing.Point(416, 40);
  this.tb_pret.Name = "tb_pret";
  this.tb_pret.Size = new System.Drawing.Size(88, 20);
  this.tb_pret.TabIndex = 2;
  this.tb_pret.Text = "tb_pret";
       //
       // btn_stg
       //
  this.btn_stg.Location = new System.Drawing.Point(56, 72);
  this.btn_stg.Name = "btn_stg";
```



```
  this.btn_stg.Size = new System.Drawing.Size(16, 23);
  this.btn_stg.TabIndex = 3;
  this.btn_stg.Text = "<";
  this.btn_stg.Click +=
            new System.EventHandler(this.btn_stg_onclick);
       //
       // btn_drt
       //
  this.btn_drt.Location = new System.Drawing.Point(80, 72);
  this.btn_drt.Name = "btn_drt";
  this.btn_drt.Size = new System.Drawing.Size(16, 23);
  this.btn_drt.TabIndex = 4;
  this.btn_drt.Text = ">";
  this.btn_drt.Click +=
            new System.EventHandler(this.btn_drt_onclick);
       //
       // Form1
       //
  this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
  this.ClientSize = new System.Drawing.Size(544, 333);
  this.Controls.Add(this.btn_drt);
  this.Controls.Add(this.btn_stg);
  this.Controls.Add(this.tb_pret);
  this.Controls.Add(this.tb_denum);
  this.Controls.Add(this.tb_cod);
  this.Name = "Form1";
  this.Text = "Exemplu legare simpla pe DataSet";
  this.ResumeLayout(false);
}
#endregion
[STAThread]
 static void Main()
   {
        Application.Run(new Form1());
   }
 }
}
```

Observaţii.

- Tabela din DataSet se poate numi altfel decât tabela din BD; i-am atribuit numele **tabela** atunci când am invocat metoda `Fill( )`, aparţinând adaptorului.
- DataBinding asigură şi conversia (`pret` este de tip `double`, iar în machetă apare ca text)

Reamintim că toate controalele fiind derivate din clasa `Control` moştenesc o colecţie numită **`DataBindings`** de tip

`ControlsBindingCollection`; ea conţine obiecte de tip `Binding` specializate în întreţinerea de legături între o proprietate a unui control şi o coloană dintr-o sursă de date.

Legătură complexă realizată vizual

Controlul de tip DataGrid se pretează cel mai bine la afişarea datelor dintr-o bază de date. Vizual, legarea unui grid la o sursă bază de date se realizează extrem de simplu.

1. Într-o aplicaţie C# Windows Application se aduce din ToolBox un control **DataGrid**, căruia i se stabileşte proprietatea **Dock** pe **Fill**, pentru a umple întreg formularul.
2. Din fereastra de Server Explorer se alege o conexiune activă şi se draghează o tabelă peste `dataGrid1`; efectul imediat constă în includerea în aplicaţie a două obiecte necesare lucrului cu baze de date: `oleDbConnection1` şi `oleDbDataAdapter1`.
3. Selectând `oleDbDataAdapter1`, cu mouse dreapta se cere **Generate DataSet**, bifând şi caseta ce solicită includerea DataSet-ului în Designer. Un obiect numit `dataSet11` este declarat automat în aplicaţie şi devine vizibil în fereastra Designer.
4. Selectând `oleDbDataAdapter1`, cu mouse dreapta se cere **Preview Data**.
5. Acum, încercând să stabilim vizual proprietatea `Data Source` a gridului, găsim în lista oferită de mediu, sursa `dataSet11`.
6. Similar, pe proprietatea `Data Member` găsim tabela `produse` a.î. să avem gridul deja configurat cu coloanele corespunzătoare.

7. Nu ne mai rămâne decât ca pe constructorul formei să cerem indirect încărcarea gridului cu date, prin intermediul DataAdaptorului:

```
this.oleDbDataAdapter1.Fill(dataSet11);
```

Dezvoltări ale acestui mecanism de legare vizuală se dau în aplicaţia din finalul acestui capitol, unde se tratează şi problema lucrului cu mai multe tabele, simultan.

Observaţii.

- La specificarea sursei de date pentru grid se precizează atât setul de date, cât şi tabela, sub forma:

```
m_dataGrid1.DataSource = ds;
m_dataGrid1.DataMember = "tbl";
```


sau se dau simultan:

```
m_dataGrid1.DataSource = ds.Tables["tbl"];
```

- Controalele ce pot conţine un şir de date, pot fi legate la o sursă după acelaşi model, atât pentru **Windows Forms**, cât şi **Web Forms**. De reţinut că pentru Windows Forms nu mai e nevoie să apelăm metoda `DataBind()` deoarece controalele UI implementează interfaţa `IEnumerable` (sau `IList`, care la rândul ei se bazează pe `IEnumerable`) şi odată ce cunosc DataSource pot să o parcurgă. Ca sursă de date pot fi folosite şi **DataTable, DataView**, vectori, colecţii etc.
- Din codul pus automat de Designer se poate observa că obiectul `BindingManagerBase` pointează spre un obiect de tip `PropertyManager`, când sursa de date returnează doar o valoare, respectiv `CurrencyManager` când sursa de date returnează o listă de valori.

2. Abordarea programatică a lucrului cu baze de date sub ADO.NET

Exerciţiu

Să se scrie o aplicaţie care efectuează operaţiile uzuale: **încărcare, adăugare, modificare** şi **ştergere**, pe tabela ***produse*** a bazei de date ***prod.mdb***. Se va folosi o machetă în care este vizibilă prin data binding o singură înregistrare din DataSet.

Rezolvare.

Aplicaţia reuneşte într-un tot unitar cunoştinţele expuse anterior, la discutarea fiecărui obiect folosit pentru accesul la baza de date, sub tehnologia ADO.NET.

Macheta cerută ar putea arăta ca în figura de mai jos, din care se poate deduce şi care sunt controalele cu care forma a fost înzestrată.

Funcţia `clickDeschide` are doar rol de testare a unei conexiuni; ea face o deschidere şi o închidere a unei conexiuni, verificând existenţa bazei de date cu care lucrează aplicaţia. Eşuarea la deschidere e semnalată printr-o excepţie adecvată; închiderea unei conexiuni este bine să se facă testând în prealabil dacă baza este deschisă în acel moment.

Se fac apoi, o deschidere şi o închidere, implicite, prin intermediul dataAdapter-ului.

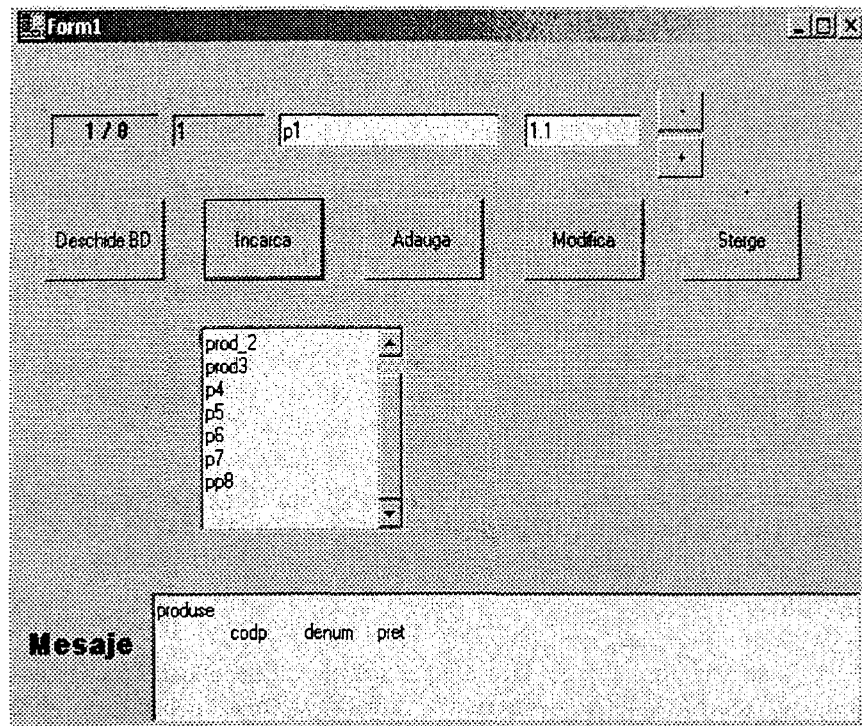


Fig. 15.1 Macheta unei aplicaţii pentru operaţiile uzuale asupra unei baze de date

```
private void clickDeschide(object sender, System.EventArgs e)
{
 string frazaSQL = "select codp, denum, pret from produse";
 string sirConex =
   "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\\prod.mdb";
 OleDbConnection conex = new OleDbConnection(sirConex);
 try { conex.Open(); mesaj.Text="BD deschisa "; }
 catch(Exception excpt)
  {
    MessageBox.Show("Esec la deschidere: "+ excpt.Message );
  }
 if (conex.State == ConnectionState.Open)
     { conex.Close(); mesaj.Text+=" si inchisa cu succes";}
 m_da= new OleDbDataAdapter(frazaSQL, conex);
 if(conex.State == ConnectionState.Closed)
     mesaj.Text+="\r\nBD redeschisa / inchisa implicit";
```



```
}
```

Funcţia **Incarca_Click** exemplifică încărcarea datelor din baza de date, folosind un DataAdapter; la încărcarea DataSet-ului se denumeşte şi tabela (tot ***produse***, ca şi în baza de date, fără a fi obligatoriu acelaşi nume), altfel am fi identificat-o numai prin **m_ds.Tables[0];** numirea putea fi făcută şi ulterior, folosind proprietatea **TableName** a tabelei.

La fiecare nouă încărcare a DataSet-ului se face o legătură simplă între trei textBox-uri şi cele trei coloane ale tabelei produse. Pentru că nu ştim în ce succesiune au fost apăsate butoanele aplicaţiei, butonul **Incarca** putând fi apăsat repetat, mai întâi eventualele colecţii de legături care există sunt şterse, apoi sunt refăcute.

```
private void Incarca_Click(object sender, System.EventArgs e)
{
 m_ds = new DataSet("setul_meu");
            // DataSet m_ds declarat la nivel de Form1

 string frazaSQL = "select codp, denum, pret from produse";
 string sirConex =
   "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\\prod.mdb";
 m_da= new OleDbDataAdapter(frazaSQL, sirConex);
 //deschidere cu sir, nu cu conexiune; m_da declarat in Form1

  m_da.Fill(m_ds,"produse");
  // m_da.Fill(m_ds);    m_ds.Tables[0].TableName="produse";

  // la apasare repetata pe Incarca trebuie sterse legaturile
  tCodp.DataBindings.Clear();
  tDenum.DataBindings.Clear();
  tPret.DataBindings.Clear();

 // refacere legaturi
  tCodp.DataBindings.Add("Text",m_ds.Tables[0],"codp" );
  tDenum.DataBindings.Add("Text",m_ds.Tables[0],"denum" );
  tPret.DataBindings.Add("Text",m_ds.Tables[0],"pret" );

nr_rec = m_ds.Tables[0].Rows.Count;
tNrCrt.Text="1 / "+nr_rec;
mesaj.Text=""; // afisare nume tabela si coloanele sale
foreach(DataTable tbl in m_ds.Tables)
 {
     mesaj.Text += tbl.TableName+ "\r\n";
     foreach(DataColumn col in m_ds.Tables[0].Columns)
           mesaj.Text+= "\t"+ col.ColumnName;
 }
string msg=""; // scrie denumirile si in textBox
foreach(DataRow linia in m_ds.Tables[0].Rows)
                              msg+=linia["denum"]+"\r\n";
```

```
 textBox1.Text= msg;
}
```

Se completează apoi câmpul de sinteză **tNrCrt**, care ne arată numărul curent al înregistrării, faţă de total înregistrări existente. În paralel în câmpul de mesaje **mesaj**, se listează structura pe coloane a tabelei *produse*. Un textBox distinct textBox1, listează denumirile produselor, exemplificând astfel accesul la liniile de date ale unui DataSet.

Funcţia de adăugare **Add_Click**, preia datele din cele trei textBox-uri şi alcătuieşte un DataRow; în esenţă, funcţia n-ar avea nevoie de existenţa DataSet-ului încărcat, dar DataRow are nevoie de o structură pe coloane; trebuie deci, fie să-i construim DataSet-ului colecţia **DataColumns Columns**, fie să facem o încărcare de DataSet, moment în care DataSet-ul va prelua structura din tabela *produse*, prin intermediul DataAdapter-ului; printr-un apel **NewRow()** structura este transferată şi noii linii de date.

```
private void Add_Click(object sender, System.EventArgs e)
{
 string sirConex =
 "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\\prod.mdb";

 DataRow lin_date=null;
 try // daca dataSet nu e incarcat ??
     {lin_date = m_ds.Tables[0].NewRow( ); }
        // [0], caci nu stie ca se numeste produse !!
 catch
  {
     m_da= new OleDbDataAdapter("select * from produse",
                              sirConex);//adaptor refolosit
     m_ds= new DataSet();
           // daca am venit direct pe Adaugare incarc m_ds
     m_da.Fill(m_ds,"produse");//conexiune implicita
     lin_date = m_ds.Tables[0].NewRow( );
  }
 if(lin_date!=null)// preluare din textBox-uri
  {
     lin_date["codp"]  = tCodp.Text; //stie de campuri!!
     lin_date["denum"] = tDenum.Text;
     lin_date["pret"]  = Convert.ToDouble(tPret.Text);
     m_ds.Tables[0].Rows.Add(lin_date); // add la dataSet
  }

 DataSet changes = m_ds.GetChanges();
  // extragem schimbarile, doar pentru a arata ca se poate
 changes.Tables[0].TableName="produse";
 mesaj.Text="Insereaza:\r\n ";
```



```
 foreach(DataRow lin in changes.Tables[0].Rows)
  {
     mesaj.Text+=lin["codp"]+"\t";
     mesaj.Text+=lin["denum"]+"\t";
     mesaj.Text+=lin["pret"]+"\r\n";
  }

 OleDbConnection myConex = new OleDbConnection(sirConex);
 string sirInsert =
  "INSERT INTO produse (codp, denum, pret)"
                + "  values (@cd,@den,@prt)";
     // luând valorile din row, are nevoie de nume simbolic
     // de parametri, pt a-i mapa pe coloanele BD
 OleDbCommand cmd = new OleDbCommand( sirInsert, myConex);
     // pregatire obiect comanda pentru Insert
 OleDbParameter param = new OleDbParameter(
                      "@cd", OleDbType.Char, 50, "codp"  );
 cmd.Parameters.Add(param);
     // adauga ceilalti parametri la variabila Command
 cmd.Parameters.Add( new OleDbParameter(
                "@den",      OleDbType.Char, 50, "denum") );
 cmd.Parameters.Add( new OleDbParameter(
                "@prt",      OleDbType.Double, 8, "pret") );

 m_da.InsertCommand = cmd;
     // echipeaza dataAdapter cu proprietatea InsertCommand

 try
  {
     m_da.Update(m_ds,"produse");
     nr_rec = m_ds.Tables[0].Rows.Count;
     BindingContext[m_ds.Tables[0]].Position = nr_rec-1;
     tNrCrt.Text = ""+
      ( BindingContext[m_ds.Tables[0]].Position+1) +" / "
                                               + nr_rec;
  }
     // acelasi lucru daca am scrie:
     //          m_da.Update(changes, "produse");
catch(Exception excpt)
  {
      MessageBox.Show("Inserare esuata: "+ excpt.Message );
  }
}
```

Linia de date pregătită în acest mod, are „marcajul" de linie nouă, astfel încât este gata pentru a fi inserată în baza de date. În scop pur didactic, se testează acest lucru, separând din DataSet-ul de bază un alt DataSet **changes**, conţinând doar liniile care au suferit schimbări, în speţă linia nou

construită. Elementele acestui DataSet sunt afişate pentru confirmare, în zona de mesaje a aplicaţiei.

Inserarea propriu-zisă se poate face numai după ce DataAdapter-ul are configurată o comandă de INSERT adecvată; în acest scop se defineşte un obiect din clasa **OleDbCommand**, i se ataşează parametrii care arată cum se mapează simbolic valorile din linia de date peste câmpurile tabelei din baza de date, după care comanda este înregistrată ca proprietate **InsertCommand** a DataAdapter-ului.

Inserarea se face uzual pe un bloc `try`, asigurându-ne de **potrivirea parametrilor cu coloanele** din tabelă, sau de **unicitatea cheii** *codp*, pentru înregistrarea nou introdusă. Concomitent, se actualizează şi afişajul de sumar, care spune ce înregistrare din noul total este acum cea vizibilă în macheta aplicaţiei.

Funcţia de ştergere **Sterge_Click** se asigură mai întâi că în momentul apăsării butonului de ştergere există stabilită o legătură între ce vedem în machetă şi o înregistrare din DataSet, altfel iese fără să facă nimic.

Când legătura există, mai cere o confirmare suplimentară a operaţiei de ştergere, după care pregăteşte comanda de ştergere, generând-o direct pe adaptor şi înzestrând-o cu cel puţin un parametru, cel invocat în comandă.

De remarcat construcţia şirului ce conţine comanda de DELETE ; ea permite furnizarea textului din textBox între apostrofuri, pentru a respecta sintaxa limbajului de interogare.

```
private void Sterge_Click(object sender, System.EventArgs e)
{
 OleDbConnection conex =new OleDbConnection(
   "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\\prod.mdb"
                                           );

if(tCodp.DataBindings.Count==0) return;
// iese cand nu are incarcat dataSet

string msg = "Stergeti produsul "+ tDenum.Text+" ?";
if( MessageBox.Show(msg,"Stergere",
        MessageBoxButtons.YesNo)==DialogResult.Yes )
 {
  m_da.DeleteCommand = new OleDbCommand(
          "DELETE from produse where denum ='"
                                 +tDenum.Text+"'", conex   );
  OleDbParameter param =new OleDbParameter(
                  "@dn", OleDbType.Char, 50, "denum"      );
  m_da.DeleteCommand.Parameters.Add(param);
```



```
  DataTable t = m_ds.Tables[0];
  DataRow [] r = t.Select("denum='"+tDenum.Text+"'");
  // intre apostrofuri, altfel il cauta ca nume de coloana

  for (int i = 0; i < r.Length; i++)
              r[i].Delete(); // marcare ca sterse, in dataSet
  int nrs = m_da.Update(m_ds,"produse");
                  // stergere si din baza de date

 if(nrs!=0)
  {
       m_ds.Clear(); // sterge sa reactualizeze conform BD
       m_da.Fill(m_ds,"produse"); // reincarca tot; merita ?
           // altfel facem doar
           // t.AcceptChanges();
           // notând astfel ca s-au operat toate modificarile
       mesaj.Text="Au fost sterse "+ nrs+ " inregistrari";
       nr_rec = m_ds.Tables[0].Rows.Count;
       tNrCrt.Text="1 / "+nr_rec;
  }
 else  mesaj.Text="Nu avem nimic de sters";
}
      // conex.Close();
}
```

Ştergerea se operează mai întâi în DataSet, pentru a marca înregistrările ca şterse; în acest scop se constituie un vector de referinţe la liniile ce vor fi afectate de ştergere (produsele cu denumirea indicată în TextBox-ul `tDenum`), după care se aplică ştergerea fiecărei linii în parte.

Acum se poate face ştergerea şi în baza de date, punând de acord liniile de date din DataSet cu cele din tabela bazei de date, prin intermediul DataAdapter-ului ; se reţine şi numărul de înregistrări şterse pentru a actualiza anunţul din zona de mesaje; totodată se şterge şi reumple DataSet-ul pentru a nota ca « originală » starea tuturor liniilor de date, efect comparabil cu cel realizat prin apelul **AcceptChanges()**.

Modificarea unei înregistrări cade în sarcina funcţiei **Modifica_Click** . Şi în acest caz ne asigurăm mai întâi că modificările din machetă s-au făcut în contextul unor legături existente între DataSet şi TextBox-urile de pe formă (adică s-a încărcat mai întâi DatSet-ul, apoi s-a selectat o înregistrare care a fost şi modificată), altfel funcţia se termină fără să facă ceva.

În contextul modificării unor înregistrări legate la machetă, modificările s-au răsfrânt deja asupra liniei din DataSet, astfel încât atribuirile din instrucţiunile comentate nu mai sunt necesare. Pot fi modificate doar câmpurile non-cheie (`denum` şi `pret`); altfel se consideră fie

că se cere modificarea altei linii decât cea vizibilă în machetă, fie negăsind nici o înregistrare cu *codp* egal cu cel modificat nu se operează modificarea.

Liniile modificate vor fi marcate ca « editate » după apelul EndEdit().

```
private void Modifica_Click
                        (object sender, System.EventArgs e)
{
 if(tCodp.DataBindings.Count==0) return;
            // iese dacă nu are incarcat dataSet

 int poz =BindingContext[m_ds.Tables[0]].Position;
 DataRow lin_date= m_ds.Tables[0].Rows[poz];
     // lin_date["denum"] = tDenum.Text;
     // lin_date["pret"]  = Convert.ToDouble(tPret.Text);

 DataTable t = m_ds.Tables[0];
 DataRow [] r = t.Select("codp='"+tCodp.Text+"'");
     // intre apostrofuri, altfel cauta ca nume de coloana

 for (int i = 0; i < r.Length; i++) r[i].EndEdit();
     //MessageBox.Show(""+r.Length+" modif in tabela");
     // DataSet changes = m_ds.GetChanges();

 OleDbConnection myConex = new OleDbConnection(
   "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=c:\\prod.mdb"
                                                     );
 string sirUpdate =
          "UPDATE produse SET denum = @den, pret = @prt " +
          "WHERE (codp = '"  + tCodp.Text+  "') ";

 OleDbCommand cmd = new OleDbCommand( sirUpdate, myConex);
     // pregatire obiect comanda de modificare
 cmd.Parameters.Add(new OleDbParameter(
          "@den",     OleDbType.Char, 50, "denum"));
 cmd.Parameters.Add(new OleDbParameter(
          "@prt",     OleDbType.Double, 8, "pret"));
 cmd.CommandText = sirUpdate;
 m_da.UpdateCommand = cmd;
 try
  {
     int nr = m_da.Update(m_ds,"produse");
     mesaj.Text="Au fost modificate "+ nr+ " inregistrari";
  }
 catch(Exception excpt)
  {
     MessageBox.Show("Modificare esuata: "+ excpt.Message );
  }
}
```


Şi în acest caz, procedura decurge tot ca şi la celelalte operaţii: pregătirea comenzii UPDATE cu parametrii ei, ataşarea ei DataAdapter-ului şi lansarea comenzii prin intermediul DataAdapter-ului. Zona de mesaje confirmă succesul sau eşecul procedurii de modificare.

3. Abordarea vizuală a lucrului cu baze de date

Cea mai mare parte a programelor de lucru cu baze de date pot fi realizate şi în manieră vizuală, folosind controale specializate din ToolBox, wizard-uri, ferestre profilate pe vizualizarea unor conexiuni sau tabele din baza de date etc.

Aplicaţia anterioară oferea o legare simplă, cu vizualizarea unei singure înregistrări. Ne propunem acum să folosim o **legare complexă** folosind un grid, care ne permite să vizualizăm toate înregistrările unui DataSet. Reamintim că metoda `Fill` a unui DataAdapter permite încărcarea unui DataSet cu toate înregistrările unei tabele din baza de date sau numai cu o parte dintre ele.

Exerciţiu

Să se scrie o **aplicaţie pentru vizualizarea şi actualizarea datelor dintr-o bază de date folosind un grid.**

Rezolvare

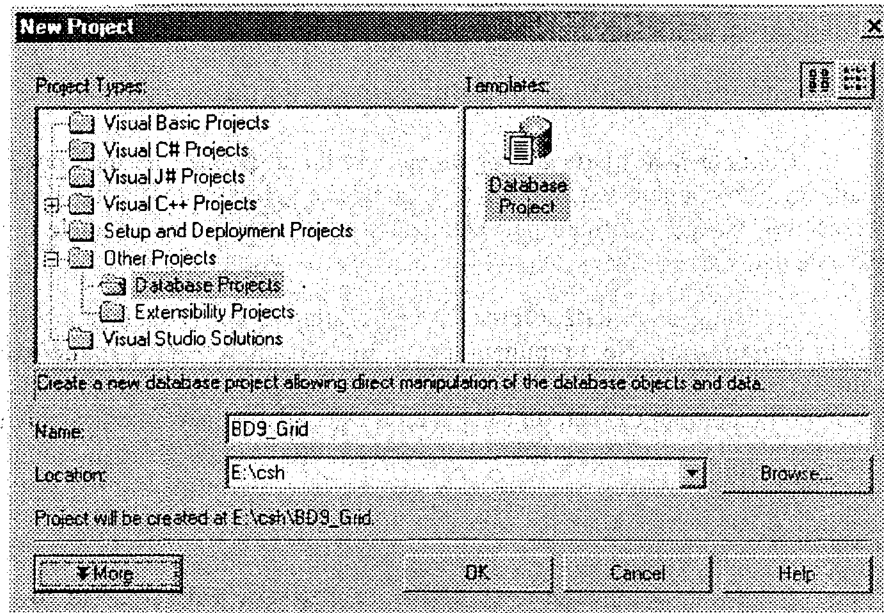
Pentru că folosirea gridului măreşte partea de vizualizare a aplicaţiei, vom folosi în acelaşi spirit, tot maniera vizuală pentru a crea cea mai mare parte a codului sursă al programului:

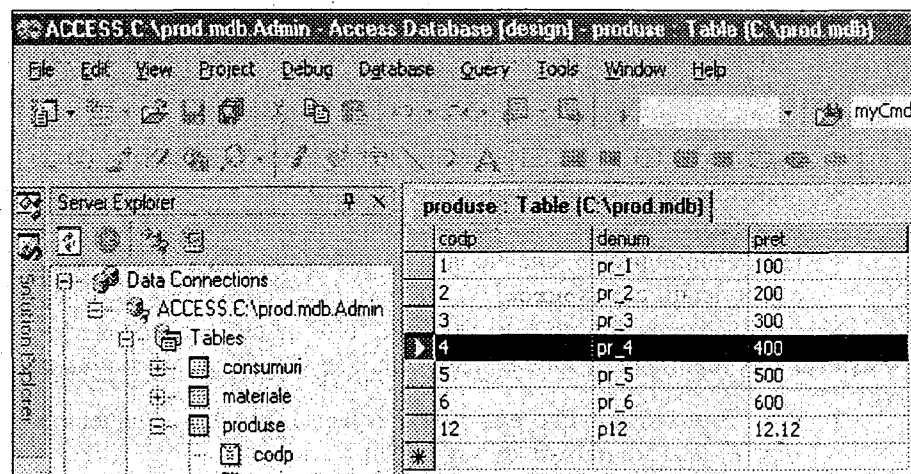
- crearea vizuală a unor conexiuni folosind **Server Explorer**;
- preluarea din **ToolBox** a obiectelor de acces la baza de date (categoria **Data**) şi explorarea vizuală a bazei de date;
- legarea unui grid de tabelele unei baze de date, prin stabilirea vizuală a proprietăţii `DataSource`;
- crearea comenzilor pe `dataAdapter`, folosind `QueryBuilder` etc.

Paşii cei mai importanţi ai aplicaţiei sunt enunţaţi şi explicaţi în continuare.

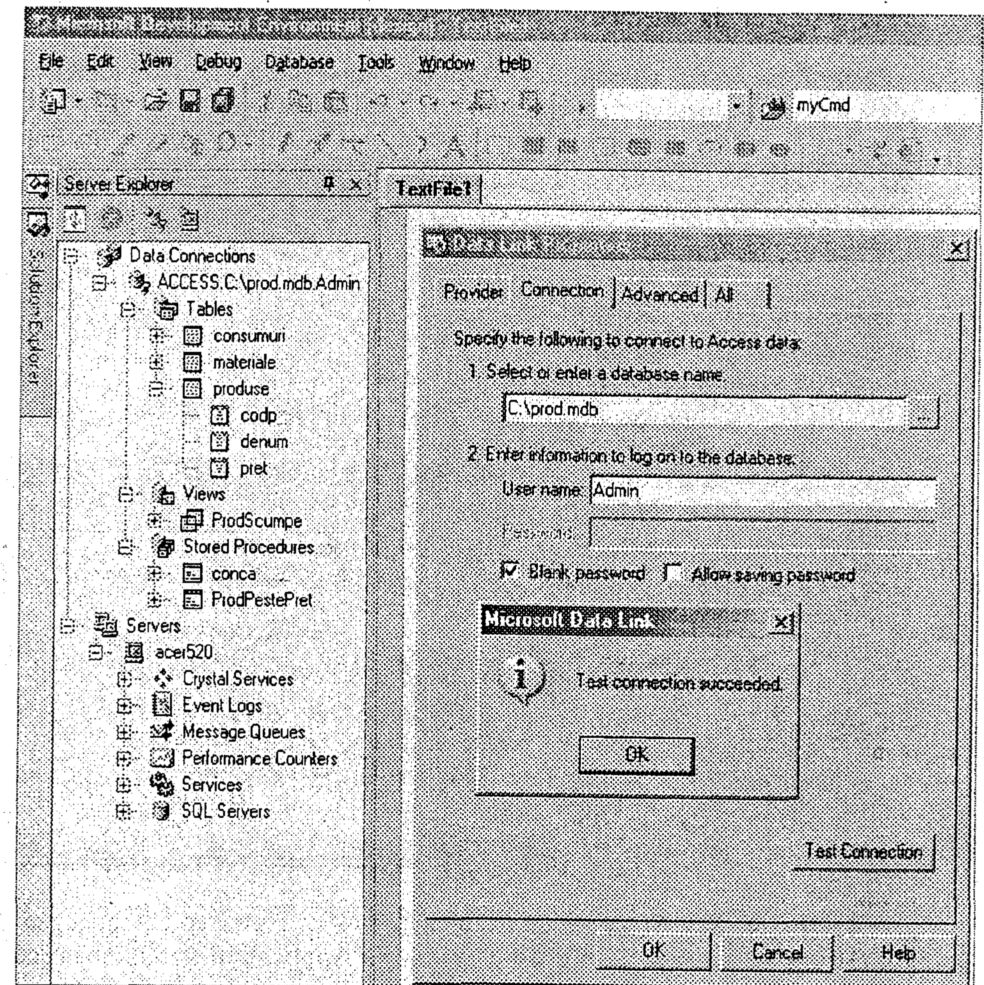
1. **File / New / Other Projects / DB Projects** (atenţie, este un template specializat, care nu este de tip C# !);

După încheierea cu succes a dialogului, în fereastra *Server Explorer* se va observa şi conexiunea nou stabilită (**ACCESS.c:\ …**) cu datele referitoare la baza de date (tabele, vizualizări, proceduri stocate etc…)



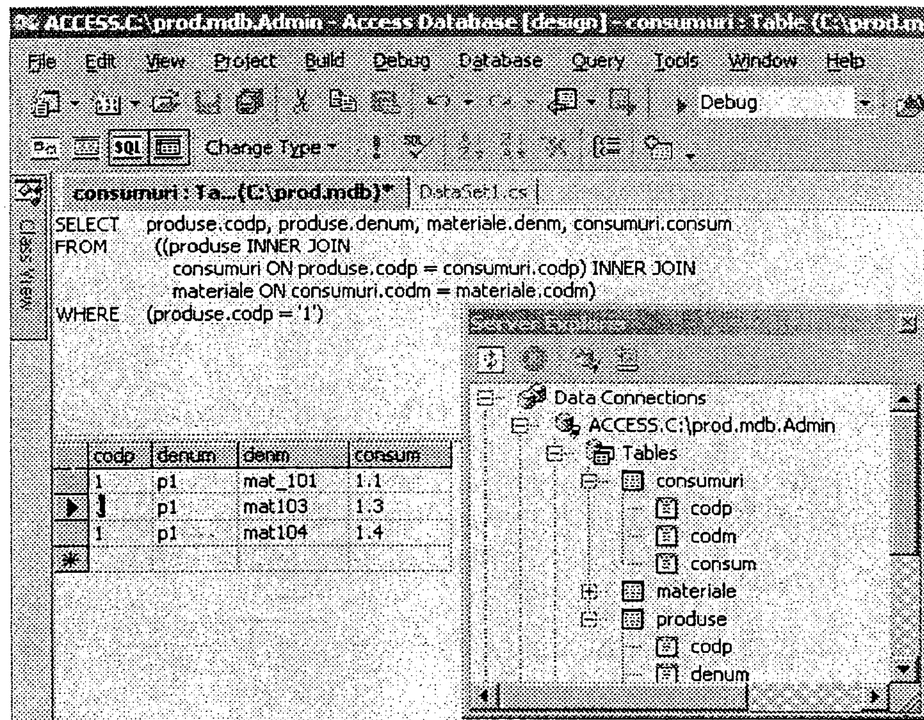


2. Cu RightClick (sau DoubleClick) pe una din tabele se poate solicita **Retrieve Data** pentru a vedea şi actualiza datele din tabelă (o deschide ca în Access, în fereastră proprie de vizualizare). Practic, acest tip de proiect îţi asigură şi modificările tipice pe o bază de date, cu salvarea noilor valori ; dispunem prin intermediul toolbar-ului sau a meniurilor, de diverse comenzi, inclusiv scriere de cod SQL.



3. Pe **DataLink Properties** (care apare automat la creare proiect sau poate fi cerut ulterior cu **Tools / Connect to DB**) se alege **MS Access Jet 4.0** ca provider şi fişierul **c:\\prod.mdb** sau alt fişier ce conţine o bază de date creată în Access şi se stabilesc proprietăţile corespunzătoare.

În SQL panel din View Designer (afişabil prin apăsarea butonului) se pot da interactiv comenzi SQL complexe, se poate vizualiza rezultatul unei cereri apăsând , sau pot fi operate modificări interactive ale datelor sau obiectelor din baza de date.



4. In Solution **Explorer** cu RightClick / **Add / New Project** adaugăm un proiect nou **de tip C# / Windows Application** şi schimbăm cu **Find / Replace All**, numele `Form1` cu **`Prel_grid`** (în numele clasei, constructor, apelul `Run()` şi numele fişierului - **`PrelGrid.cs`**). Se observă că pe aceeaşi Solution putem lucra cu mai multe proiecte !

5. Cu drag & drop se aduc din Server Explorer tabelele (ex. tabela **produse**) sau views-urile dorite peste forma din [Design]; operaţiunea are ca efect crearea a două obiecte **`oleDbConnection1`** şi **`oleDbDataAdapter1`**. În plus, se crează versiuni implicite ale celor patru comenzi (SELECT, INSERT, DELETE, UPDATE) ale dataAdapter-ului, cu parametrii şi direcţia lor, inclusiv schema de mapare (**`TableMappings`**) a parametrilor peste câmpurile corespunzătoare din tabela bazei de date.

Aşadar, `oleDbConnection1` conţine deja informaţiile despre *provider*, baza de date, string-ul de conectare ; `oleDbDataAdapter1` conţine deja variante iniţiale pentru comenzile de **`Select`**, **`Update`**, **`Delete`**, **`Insert`**, modificabile dacă selectăm `oleDbDataAdapter1` şi în Properties pun alt `Name`, `Text` etc.

RightClick pe `oleDbDataAdapter1` oferă deja trei facilităţi:

a) **Configure Data Adapter** va lansa un wizard, dacă dorim să modificăm, pe paşi, proprietăţile dataAdapter-ului, în speţă comenzile de `Select`, `Update` etc. Se poate opta pentru QueryBuilder să configurăm vizual comenzile; acelaşi lucru se poate face şi cu Properties asociat obiectului DataAdapter, alegând o comandă (ex. Select), iar pe CommandText, buton "... " apare QueryBuilder cu care configurăm fraza select.

b) **Generate Data Set** pentru a pune condiţii pe setul de date selectate;

c) **Preview Data** pentru a vizualiza datele selectate.

6. Generăm cu RightClick **Generate Data Set** şi se crează o clasă **`DataSet1`** şi un obiect set de date **`dataSet11`** pe care îl lăsăm cu valorile implicite; în Solution Explorer apare un fişier **`dataSet1.xsd`**, pe care dacă dăm click vedem descrierea câmpurilor preluate.

7. Se inserează din ToolBox un obiect **DataGrid**; îi punem proprietatea **`Layout / Dock`** pe mijloc (**`Fill`**, adică umple întreaga fereastră formular) şi proprietatea **`Data Source`** pe **`dataSet1`**. Pe proprietatea **`Data Member`** selectăm tabela ***produse*** a.î. la rulare să avem gridul deja expandat. Se pot pune şi alte proprietăţi, dar mai puţin importante : **`CaptionText`**, **`CaptionVisible`** etc. Se putea alege ca proprietate **`Data Source`** direct obiectul **`dataSet11`**.

8. Pe constructorul formei se cere încărcarea setului de date în grid:
```
oleDbDataAdapter1.Fill(dataSet11);
```

9. Dacă a eşuat crearea implicită a comenzilor Insert/Delete/Update (vezi pas 5) creăm noi aceste comenzi punându-le cu editorul pe proprietăţile **InsertCommand**, **DeleteCommand** şi respectiv **UpdateCommand** ale obiectului `oleDbDataAdapter1`: mai întâi fixăm `Connection` pe conexiunea existentă, apoi pe `CommandText` cu QueryBuilder configurăm comenzile astfel:

INSERT INTO produse (codp, denum, pret) VALUES (?, ?, ?)
DELETE FROM produse WHERE (codp = ?)
UPDATE produse SET denum = ?, pret = ? WHERE (codp = ?)

10. Adăugăm un meniu (din Toolbox) cu opţiunile : **Salvare** şi **Iesire**. Pe evenimentul click al opţiunii de **Salvare** punem o funcţie cu codul:
```
try
{
 DataSet schimbat = dataSet11.GetChanges();
 if(schimbat!=null)
  {
     int nlin =oleDbDataAdapter1.Update(schimbat);
     dataSet11.AcceptChanges();
     MessageBox.Show("Salvat "+nlin+" linii de date");
  }
 else
     {MessageBox.Show("Nimic de salvat !");}
}
catch (Exception eroare)
{
     MessageBox.Show("Eroarea:  "+ eroare.Message );
     dataSet11.RejectChanges();
}
```

11. Pe evenimentul click al opţiunii **Iesire** punem o funcţie cu codul:
```
Application.Exit();
```

Putem acum testa, la rulare inserând, modificând sau ştergând una sau mai multe înregistrări şi **mutăm mouse-ul pe o altă linie**; la opţiunea `Salvare`, ni se afişează câte linii au fost salvate sau eventualele erori de salvare.

Aducând în grid mai multe tabele între care există deja stabilite relaţii, gridul oferă posibilitatea navigării în profunzime, pe aceste relaţii.

Lucru în grid cu DataSet-uri cu mai multe tabele de date corelate

Ne punem problema cât de uşor putem programa **vizual**, folosind un grid în care să vedem date provenind din mai multe tabele, corelate între ele. Să presupunem că dorim să scriem un program de actualizare a consumurilor specifice. Operatorul nu se poate descurca dacă lucrează doar pe tabela de consumuri, pentru că nu ştie pe de rost coduri de produse şi de materiale pentru a modifica în cunoştinţă de cauză un consum specific. Trebuie aşadar să aducem în grid şi denumirile în clar, preluate din tabelele produse şi materiale.

Pentru aceasta nu avem decât ca într-o aplicaţie Windows C# simplă să parcurgem paşii următori:

1. inserăm un **DataGrid**
2. dragăm o tabelă dintr-o bază de date vizibilă în **Server Explorer**
3. generăm un dataSet selectând **oleDbDataAdapter1** şi folosind mouse RightClick
4. legăm gridul de **dataSet1** prin intermediul proprietăţii **DataSource** a gridului
5. cu **oleDbDataAdapter1** selectat, modificăm proprietatea **Select**:

```
SELECT   produse.denum,   materiale.denm,   consumuri.codp,
consumuri.codm, consumuri.consum
FROM  ((produse INNER JOIN consumuri
        ON produse.codp = consumuri.codp )
            INNER JOIN materiale
                ON consumuri.codm = materiale.codm )
ORDER BY produse.denum
```

6. Pe constructor sau pe `Load_Form` se cere încărcarea DataSet-ului:

```
oleDbDataAdapter1.Fill(dataSet11,"mixta");
```

7. Opţiune `Save` a meniului va conţine acum codul:

```
private void Save_Click(object sender, System.EventArgs e)
{
	try
	{
		oleDbDataAdapter1.Update(dataSet11,"mixta");
	}
	catch( Exception exc)
	{
		MessageBox.Show("Esec update "+exc.Message);
	}
}
```

8. Pe proprietăţile obiectului `oleDbDataAdapter1`, modificăm comanda de **`Update`** pentru a opera doar pe consumuri, modificându-le conform modificărilor făcute de utilizator în grid :

```
UPDATE    consumuri
SET              codp = ?, codm = ?, consum = ?
WHERE     (codp = ?) AND (codm = ?)
```

9. După aceasta se regenerează de fiecare dată **`dataSet11`** (**`oleDbDataAdapter1`** selectat şi MouseRight / **Generate DataSet**).

Se observă că în astfel de situaţii mediul nu mai ştie să genereze automat comenzile de lucru cu baza de date. Putem da comanda select într-o formă primară, mult mai apropiată de utilizator:

```
SELECT   produse.denum, materiale.denm,  consumuri.consum
FROM    produse, materiale, consumuri
WHERE produse.codp   = consumuri.codp AND
      consumuri.codm = materiale.codm
ORDER BY produse.denum
```

dar ea va fi înlocuită automat, la compilare, cu forma standard cu INNER JOIN, aşa cum apare în secvenţa de program de la pasul 5.

CONTROALE DE UTILIZATOR

1. **Construirea şi testarea controalelor de utilizator**
2. **Construirea bibliotecilor de componente utilizator**

1. Construirea şi testarea controalelor de utilizator

Prin **control de utilizator** înţelegem o componentă cu o anumită funcţionalitate construită de programator şi bazată uzual pe o formă, stocată într-o bibliotecă dinamică (dll), care poate fi inserată în ToolBox de unde poate fi preluată cu uşurinţă şi folosită în diverse alte aplicaţii.

Exerciţiu

Să se creeze un control de utilizator « Ceas programabil » care oferă următoarea funcţionalitate:

- poate fi programat să sune la o anumită oră şi minut;
- poate bloca tastatura, dacă la momentul fixării cuantei de timp cronometrate s-a optat pentru această opţiune.

Să se testeze un astfel de control într-o aplicaţie de examinare, bazată pe test grilă.

Rezolvare

1. **File / New / Project** şi se alege **Windows Control Library** ca template de aplicaţie. Apare o formă cu nume implicit **`UserControl1`**, pe care îl putem schimba după dorinţă; noi cu **Find / Replace** vom schimba **`UserControl1`** cu **`CeasCtrl`** şi pentru a o referi uşor în continuare şi pentru a o distinge mai departe de forma principală a aplicaţiei client, ce foloseşte ceasul drept control.

   Preferabil să alegem pentru fereastra viitorului control un stil simplu: `FormBorderStyle SizableToolWindow`.

2. Prin aducerea unui control **Timer** din ToolBox, sau scriind direct cod sursă, realizăm instanţierea unui obiect Timer, căruia i se stabilesc principalele proprietăţi (`Interval 1000`, intervalul la care să ticăie – o secundă, `Enabled true`, adică timer activat); dacă am lucrat vizual (cu designer-ul), în acest moment avem:
   - în definiţia clasei **`CeasCtrl`**, declaraţia:
     `private System.Windows.Forms.Timer timer1;`

- iar în `InitializeComponent()`:

```
this.timer1 =
    new System.Windows.Forms.Timer(this.components);
this.timer1.Enabled = true;
this.timer1.Interval = 1000;
// this.timer1.Start();

this.Name = "CeasCtrl";
```

3. Cu controlul **`timer1`** selectat, se crează cu **Properties** / secțiunea **Events** și apoi se înregistrează automat **funcția de tratare a evenimentului Tick**:

```
this.timer1.Tick +=
        new System.EventHandler(this.timer1_Tick);
```

4. Cu forma **`CeasCtrl`** selectată, se crează cu **Properties** / secțiunea **Events** și apoi se înregistrează **funcția de tratare a evenimentului** **`Paint`** al formei, pentru că avem nevoie ca la fiecare secundă să se retraseze ceasul, cu poziția acelor actualizată.

```
this.Paint +=
    new PaintEventHandler(this. CeasCtrl_Paint);
```

5. Declarăm forma drept retrasabilă la redimensionare, punând în constructorul ei proprietatea **`ResizeRedraw`** pe `true`:

```
    this.ResizeRedraw = true;
```

6. Se adaugă în definiția clasei **`CeasCtrl`** trei variabile ce vor ține coordonatele temporale:

```
public int ora,min,sec;
```

7. Se scrie efectiv funcția de tratare a evenimentul **`Paint`**, astfel încât să traseze secundarul, minutarul și orarul, la un unghi proporțional cu secunda, minutul și ora exactă :

```
private void CeasCtrl_Paint
      (object sender, System.Windows.Forms.PaintEventArgs e)
{
  int i;
  SolidBrush
      pnsRosie =
            new SolidBrush(Color.FromArgb(120, 255, 0, 0)),
      pnsNeagra =
            new SolidBrush(Color.FromArgb(120, 0, 0, 0)),
```



```
      pnsAlbastra =
            new SolidBrush(Color.FromArgb(120, 0, 0, 255) );

  Pen   penSec = new Pen(pnsNeagra,1),
        penMin = new Pen(pnsAlbastra,2),
        penOra = new Pen(pnsRosie,4)       ;

  Pen pen = new Pen(pnsRosie,3);
  Rectangle rect = e.ClipRectangle;
  rect.X+=5; rect.Y+=5; rect.Height-=10; rect.Width-=10;
                // dreptunghi de vizualizare
  int Raza = rect.Height/2-10;
  Graphics g = e.Graphics;
  g.DrawEllipse(pen,rect);
  int x0 = rect.X+= rect.Width/2;
  int y0=rect.Y+=rect.Height/2;
  cerculet(x0,y0,5,g);

  for(i=0; i<12; i++) // cadranul
     {
          int x, y;
          x=x0+ (int)(Raza*Math.Cos(i*Math.PI/6.0));
          y=y0-(int)(Raza*Math.Sin(i*Math.PI/6.0));
          patratel(x,y,8,g);
     }

  Point     centru = new Point(x0,y0),
            capSec = new Point(
        x0+ (int)(Raza*Math.Cos(Math.PI/2-sec*Math.PI/30.0))
        y0-(int)(Raza*Math.Sin(Math.PI/2-sec*Math.PI/30.0))
                                 );
  g.DrawLine(penSec,centru,capSec);

  Point capMin = new Point(
     x0+ (int)((Raza-15)*Math.Cos(Math.PI/2-
                     (min+sec/60.0)*Math.PI/30.0)),
     y0- (int)((Raza-15)*Math.Sin(Math.PI/2-
                     (min+sec/60.0)*Math.PI/30.0))
     );
  g.DrawLine(penMin,centru,capMin);

  Point capOra=new Point(
     x0+ (int)((Raza-30)*Math.Cos(Math.PI/2-
                     (ora+min/60.0)*Math.PI/6.0)),
     y0- (int)((Raza-30)*Math.Sin(Math.PI/2-
                    (ora+min/60.0) *Math.PI/6.0)) );
  g.DrawLine(penOra,centru,capOra);
}
```

Se completează clasa **`CeasCtrl`** cu două funcții auxiliare, necesare trasării cadranului ceasului:

```
void cerculet(int x, int y, int raza, Graphics g)
{     // cerculet cu centru si raza date
      Rectangle r = new Rectangle
```

```
            (new Point(x-raza,y-raza), new Size(2*raza,2*raza));
        Brush pens = new SolidBrush(Color.Red);
        g.FillEllipse(pens,r);
}

void patratel(int x, int y, int latura, Graphics g)
{       // patratel dat prin centru si latura sa
        Rectangle r = new Rectangle(
              new Point(x-latura/2,y-latura/2),
              new Size(latura,latura)         );
        SolidBrush pns =
              new SolidBrush(Color.FromArgb(120, 0, 0, 255));
        Pen pen = new Pen(pns,2);
        g.DrawRectangle(pen,r);
}
```

8. Se completează funcţia de tratare a evenimentului Tick astfel încât după preluarea timpului ca `string` să extragă ora, minutul şi secunda şi să le stocheze ca întregi, în variabilele alocate anterior la nivelul formei:

```
private void timer1_Tick(object sender, System.EventArgs e)
{
        string strOra = DateTime.Now.ToLongTimeString();
        // similar cu a prelua direct DateTime.Now.Hour ...
        if(strOra[1]==':')
        {
              ora = Convert.ToInt32(strOra.Substring(0,1)  );
              min = Convert.ToInt32(strOra.Substring(2,2));
              sec=Convert.ToInt32(strOra.Substring(5,2));
        }
        else
        {
              ora = Convert.ToInt32(strOra.Substring(0,2));
              min = Convert.ToInt32(strOra.Substring(3,2));
              sec=Convert.ToInt32(strOra.Substring(6,2));
        }
        this.Invalidate();
}
```

9. Se adaugă în clasa **CeasCtrl** două variabile ce ţin ora şi minutul la care este programat ceasul să sune:

```
    private int sunaOra, sunaMin;
```

pe care le iniţializăm cu 0, în constructorul clasei.

10. Adăugarea proprietăţilor **SunaLaOra** şi **SunaLaMin** controlului de tip ceas; prin acestea, **aplicaţia client** va putea fixa ora şi minutul la care doreşte să sune ceasul.
În **ClassView** RightClick pe forma **CeasCtrl**; alegem **Add / Add Property** şi completăm numele proprietăţilor şi conţinutul accesorilor, încât să arate astfel:

```
    public int SunaLaOra
            {
                  get   {     return sunaOra;    }
                  set   {     sunaOra = value;   }
            }

    public int SunaLaMin
            {
                  get   {     return sunaMin;    }
                  set   {     sunaMin = value;   }
            }
[Browsable(true),
EditorBrowsable(EditorBrowsableState.Never),
Category("Custom")]
```

Atributul `EditorBrowsableState.Never` de mai sus se adaugă doar dacă vrem ca proprietatea să nu fie vizibilă în editorul de proprietăţi.

Nu trebuie să ne impacientăm că nu vedem încă proprietăţile controlului ceas în fereastra de proprietăţi. În editor nu se văd proprietăţile decât pentru clasele terminate; adică putem vedea doar proprietăţile unei clase de bază, nu ale celei pe care o derivăm acum şi este deocamdată în construcţie. Vom vedea proprietăţile acesteia abia când o vom folosi într-o aplicaţie client. Spre exemplu, în orice aplicaţie vedem proprietăţile formei, dar este vorba de clasa de bază `Form`, nu de cea derivată de noi din `Form` şi denumită implicit `Form1`.

Se denumesc prin **proprietăţi de ambient** (de exemplu `BackColor`, `ForeColor`, `Font`) proprietăţile ale căror valori dacă nu le definim noi, ele împrumută valorile proprietăţilor echivalente ale containerului (ambientului).

11. Creare aplicaţiei de testare a controlului de utilizator

Încă de la început, observăm că aplicaţia se compilează, dar nu rulează direct, ci avem nevoie de o aplicaţie care să folosească acest control.

Solution permite să avem în acelaşi timp mai multe proiecte deschise într-o sesiune; putem deci lucra concomitent pe controlul nostru şi verifica efectul modificărilor în aplicaţia care îl va utiliza. *Solution* permite aşadar şi depanarea multiproiect.

- Meniu **File , Add + New Project** şi se alege de această dată, **C# Windows Application** pe care îl vom denumi **TestareCeas;**
- meniu **Project / Set as Startup Project**; stabileşte că acest proiect se va lansa primul în cadrul acestei soluţii; în felul acesta putem lucra încă pe controlul de utilizator, iar mediul va lansa mereu aplicaţia de test cu noua variantă a controlului, deja înglobată.

12. Adăugarea controlului de tip ceas în ToolBox-ul aplicaţiei client.

În noul proiect, pe meniu **Tools / Add Remove ToolBox Items** putem adăuga Framework Components, COM-uri sau alegând **Browse** căutăm directorul care conţine Ceas.dll, creat anterior; după inserare vom găsi deja în ToolBox, de obicei în secţiunea My User Controls, controlul **CeasCtrl** creat de noi anterior.

13. Aducerea controlului pe forma TestareCeas prin tragere din ToolBox ne dezvăluie un lucru extrem de important: prin inserarea controlului, **acesta este instanţiat şi începe să funcţioneze**, chiar dacă aplicaţia client nu este încă în execuţie. Acest lucru este carcteristic componentelor, ele având existenţă de sine stătătoare, dialogând cu aplicaţia client ca de la executabil la executabil.

14. Adăugarea unui eveniment pe controlul de utilizator

- În partea de început a clasei `CeasCtrl`, unde sunt declaraţi şi ceilalţi membri ai clasei se scrie:

```
public event System.EventHandler Suna;
```

- La rulare, se vede deja în fereastra Properties / Events **a clasei client, selectând controlul** `CeasCtrl` tras din ToolBox, evenimentul numit `Suna`, iar aplicaţia de test poate subscrie la acest eveniment.

15. Subscrierea formei client la eveniment prin înregistrarea unei funcţii de tratare a mesajului emis de evenimentul `Suna` se poate face în manieră vizuală sau scriind codul sursă. Putem da *double click* pe acest eveniment şi se crează automat funcţia :


```
private void ceasCtrl1_Suna
        (object sender, System.EventArgs e) {     }
```

O vom completa pentru simplitate, să scrie timp de un minut textul « Suna ceasul » într-un textBox cu proprietatea `Multiline` pe `true`, adăugat în prealabil pe forma aplicaţiei client :

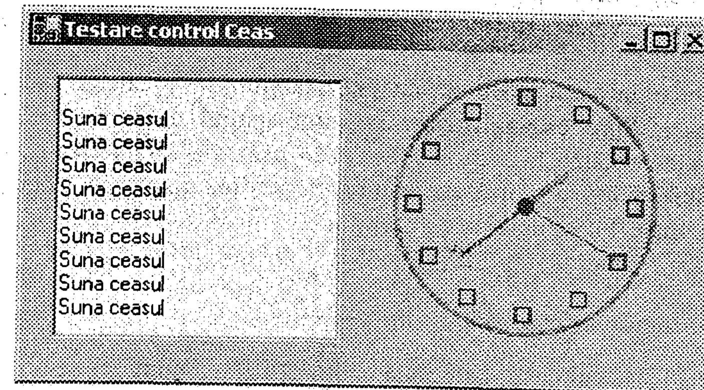
```
private void ceasCtrl1_Suna
        (object sender, System.EventArgs e)
{
    textBox1.Text+="\r\nSuna ceasul";
}
```

Singura problemă rămasă este cine declanşează evenimentul `Suna` ; pentru aceasta am adăugat în finalul funcţiei de tratare a evenimentului `Tick` în `CeasCtrl` apelul:

```
if(ora==this.sunaOra && Suna!=null )
        if(min==this.SunaLaMin);
                    Suna(this,new EventArgs());
```

Lansarea evenimentului se face doar dacă delegatul conţine vreo metodă ataşată de aplicaţia client (`Suna!=null`); altfel, controlul static din aplicaţia client rulând încă înainte de a rula aplicaţia client, va declanşa evenimentul, dar delegatul neavând ataşată încă metodă va da excepţie de referinţă nulă.

Dacă înainte de a lansa aplicaţia client avem grijă să fixăm controlului ceas proprietăţile `SunaLaOra` şi `SunaLaMin` pe valorile 13 şi 38 sau 1 şi 38, adică în format lung sau scurt după cum este setat să afişeze şi ceasul din Windows şi apoi lansăm aplicaţia, la ora şi minutul respectiv ceasul va suna !



Pentru a răspunde ultimei cerințe, legată de blocarea tastaturii la atingerea momentului critic vă sugerăm folosirea celor discutate în capitolul Gestiunea evenimentelor generate de tastatură , în speță `e.Handled=true;` din blocul de parametri ai evenimentului `KeyPress`.

2. Construirea bibliotecilor de componente utilizator

Am văzut deja cum se construiește o componentă de utilizator și cum este ea integrată într-o aplicație client. În mare parte lucru s-a desfășurat în manieră vizuală, iar componenta de utilizator a fost înzestrată cu interfață vizuală, pentru a fi ușor programabilă și manipulabilă.

În principiu, este posibilă și crearea de componente fără interfață vizuală, stocabile tot în biblioteci dinamice și care oferă clase și funcții cu oarecare generalitate, utile în alte aplicații. Pentru a putea face comparații, am realizat aproximativ aceeași componentă și în versiunea cu interfață vizuală, subliniind astfel mai bine diferențele.

Componente fără interfață vizuală

Vă prezentăm modul în care programatorul poate să-și construiască un **domeniu de nume** (`namespace`) în care să-și definească clase, pe care ulterior le va folosi în alte aplicații.

I. Construirea bibliotecii de clase presupune efectuarea următorilor pași:

1. Se construiește un nou proiect **File / New / Project** și se va alege la *Project Types* **Visual C# Project**, iar la *Templates* **Class Library**, numele dat de programator fiind `Bib1`; (a se observa că template-ul diferă de cazul anterior, când era **Windows Control Library**).

2. Pentru implementare trebuie să stabilim în prealabil funcționalitatea clasei; Spre exemplificare, definim spațiul de nume `Statistic` și implementăm clasa `Colectivitate` pentru a caracteriza o colectivitate statistică cu ajutorul indicatorilor: **medie, dispersie** și **coeficient de variație.** Formulele de calcul sunt:

- **media aritmetică:** $m = \frac{\sum_{i=1}^{n} x_i}{n}$,

  unde x_i reprezinta observațiile și n numărul de observații;

- **dispersia:** $\sigma = \frac{\sum_{i=1}^{n} (x_i - m)^2}{n}$;

- **coeficientul de variație:** $cv = \frac{\sqrt{\sigma}}{m}$.

Vom completa fișierul `Bib1.cs` după cum urmează:

```
using System;
namespace Statistic
{
 public class Colectivitate
 {
  //sectiunea privata
  int no; // numarul de observatii
  bool vb; // flag - daca indicatorii sunt calculati
  int[] vo = new int[100]; // vectorul de observatii
  double m,d,cv; // medie, dispersie si coef. de variatie

  //calcul medie
  void c_med()
   {
      double s=0.0;
      for(int i=0; i<no; i++) s+=vo[i];
      try
      {
            if(no==0) throw new Exception();
            m=s/no;
      }
      catch(Exception)
      {
            Console.WriteLine("\n\tImpartire la ZERO!!
                  \n\tColectivitate fara observatii!!");
            Console.Read();
            Environment.Exit(1);
      }
   }

 //calcul dispersie
 void c_disp()
 {
     double s=0.0;
     for(int i=0; i<no; i++) s+=(vo[i]-m)*(vo[i]-m);
     d=s/no;
 }

 //calcul coeficient de variatie
```

```
void c_cv()            {     cv=Math.Sqrt(d)/m;       }

//calcul indicatori
void calcul()    { c_med(); c_disp(); c_cv();  }

//sectiunea publica
//constructori
public Colectivitate()        { no=0; vb=true; }
public Colectivitate(int k)  { no=k; vb=true; }

//adaugarea unei observatii la colectivitate
public void Add(int z)        {  vo[no++]=z; vb=true;    }

//citirea de la tastatura a observatiilor
public void citire()
{
    for(int i=0;i<no; i++)
    {
        Console.Write("Observatia {0} =",i+1);
        vo[i]=Convert.ToInt32(Console.ReadLine());
    }
}

//conversia rezultatelor la sir
public override string ToString()
{
    if(vb){ calcul(); vb=false; }
        return " Media="+m.ToString("F")+"
                    Dispersia="+d.ToString("F")+
        " Coeficient de variatie="+cv.ToString("F");
}

//proprietati
public double media  //media - read onlz (calculat)
    {
        get
        {
            if(vb){ calcul(); vb=false; }
            return m;
        }
    }

public double dispersia //dispersia - read only (calculat)
    {
        get
        {
            if(vb){ calcul(); vb=false; }
            return d;
        }
    }
```



```
    public double coef_variatie //read only (calculat)
        {
            get
            {
                if(vb){ calcul(); vb=false; }
                return cv;
            }
        }

    //accesul la observatii prin indexare
    public int this [int i]
        {
            get
            {
                if (i < 0 || i >= no) return 0;
                else  return vo[i];
            }
            set
            {
                if (!(i < 0 || i >= no))
                    {vb=true; vo[i] = value;}
            }
        }
    }
}
```

Prin opțiunile meniului **Build / Build Solution** se obține fișierul `bib1.dll` care va conține biblioteca de clase.

II. Utilizarea claselor din biblioteci în aplicații independente presupune efectuarea pașilor următori:

1. Construirea unui nou proiect **File / New / Project** pentru care se va alege la *Project Types* **Visual C# Project,** iar la *Templates* **Console Application,** numele fiind `TestBib`;
2. Se vizualizează fereastra *Solution Explorer* (**View / Solution Explorer**);
3. Se deschide controlul de tip *tree* care are rădăcina `TestBib` și o ramură `References`;
4. Se selectează `References`, se dă click pe butonul dreapta mouse și se alege **Add References;**
5. In căsuța de dialog se selectează **Projects** după care se apasă pe butonul **Browse** și se alege fișierul `Bib1.dll` care conține biblioteca dorită; în urma operației, la `References` va apare și `Bib1`;
6. In program, înainte de a folosi, clasa `Colectivitate`, trebuie să se invoce spațiul de nume `Statistic`:
    `using Statistic;`

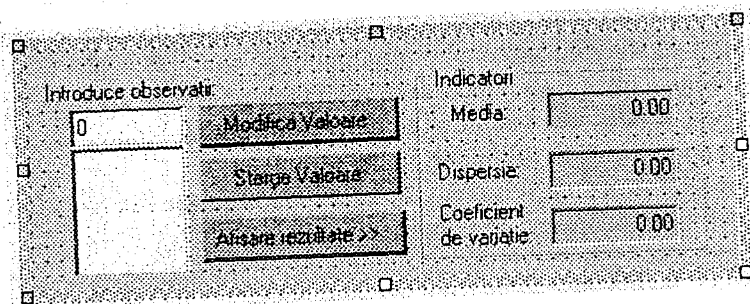
Programul care utilizează biblioteca creată anterior poate avea structura următoare:

```
using System;
using Statistic;
class Principal
{
static void Main(string[] args)
{
        Colectivitate c1 = new Colectivitate(3),
                          c2 = new Colectivitate();
        c1.citire();
        Console.Write("Rezultate: {0}",c1);
        Console.Write("\n Adaugati o noua observatie:");
        int x=Convert.ToInt32(Console.ReadLine());
        c1.Add(x);
        Console.Write("Noile rezultate: {0}",c1);
        Console.Write
              ("\nObservatie 1: {0} Modificati val:",c1[0]);
        x=Convert.ToInt32(Console.ReadLine());
        c1[0]=x;
        Console.Write("Media recalculata: {0}",c1.media);
        Console.Write
              ("\nRezultate colectivitate fara obs:{0}",c2);
        Console.Read();
        }
}
```

Componente cu interfaţă vizuală

I. Crearea unei componente de utilizator

Tot în vederea caracterizării unei colectivităţi statistice cu ajutorul indicatorilor medie, dispersie şi coeficient de variaţie, vom realiza o componentă cu interfaţă vizuală ca în figura următoare. Ca modalitate vom folosi aceeaşi tehnică folosită şi pentru componenta CeasCtrl, creată anterior; micile deosebiri arată diverse alternative pentru a realiza acelaşi lucru.



1. Se construieşte un nou proiect **File / New / Project** şi se va alege la *Project Types* **Visual C# Project**, iar la *Templates* **Windows Control Library,** numele fiind **`comp_viz;`**
2. Se generează, pentru a implementa operaţiile ce ţin de interfaţă, o clasă denumită în cazul nostru **`Indicatori`**, după cum urmează:

```
public class Indicatori : System.Windows.Forms.UserControl
```

3. Se va completa vizual forma, care va constitui interfaţa vizuală, cu controalele, ca în figura amintită;
4. Se adaugă la proiect o nouă clasă (**Project / Add Class**), numele fiind **`Colectivitate`**, pentru a implementa efectiv partea de calcul a indicatorilor (clasă este descrisă ca în exemplul anterior).

Pe forma componentei există controale cu rol în introducerea, modificarea respectiv ştergerea observaţiilor şi controale de tip **TextBox** cu rol în afişarea rezultatelor. Controalelor de tip TextBox, pentru a nu le permite editarea valorii, li s-a asociat proprietăţii **`ReadOnly`** valoarea **`True`**.

- **Introducerea datelor** (observaţiilor) se realizează printr-un control de tip ComboBox (**cB1**) pentru care proprietatea de **`DropDownList`** este **`Simple`**. In casuţa de editare se vor introduce valori care vor fi adăugate în listă prin apăsarea tastei ENTER; acest lucru se face răspunzînd la mesajul **`KeyUp`**:

```
private void cB1_KeyUp(object sender, KeyEventArgs e)
 {
  if (e.KeyData==Keys.Enter)
   {
      cB1.Items.Add(cB1.Text); // adaugare text in combo
      cB1.Text="0";            // reinitializare zona editare
      cB1.Select(0,1);         // selectie continut zona
                                                    //de editare
   }
 }
```

- **Modificarea** valorii unei observaţii presupune mai întîi selectarea ei din lista de observaţii; se răspunde la mesajul **`SelectedIndexChanged`** pentru controlul de tip comboBox:

```
private void cB1_SelectedIndexChanged
                          (object sender, System.EventArgs e)
    {
          idx = cB1.SelectedIndex;
    }
```

Variabila **idx** de tip întreg este membră a clasei **Indicatori** şi reţine indexul observaţiei selectate din controlul de tip ComboBox.
Valoarea selectată va apare în zona de editare a controlului de tip ComboBox, se editează corespunzător după care se apasă pe butonul inscripţionat **Modifica Valoare**, care declanşează funcţia:

```
private void button1_Click(object sender, System.EventArgs e)
{
 if(idx!=-1)
      {
            string sir=cB1.Text;
            cB1.Items[idx]=sir;
            cB1.Focus();
      }
 else
      MessageBox.Show(
        "Nu este selectata valoare pentru modificare!",
        "Mesaj de eroare",MessageBoxButtons.OK,
         MessageBoxIcon.Warning     );
}
```

Ştergerea valorii unei observaţii presupune mai întîi selectarea ei, după care se apasă butonul inscripţionat **Sterge Valoare**:

```
private void button3_Click(object sender, System.EventArgs e)
{
 idx=cB1.SelectedIndex;
 if(idx!=-1)
      {
            cB1.Items.RemoveAt(idx);
            cB1.Text="0";
            cB1.Select(0,1);
      }
 else
      MessageBox.Show(
         "Nu este selectata valoare pentru stergere!",
         "Mesaj de eroare",MessageBoxButtons.OK,
          MessageBoxIcon.Warning     );
}
```

Afişarea rezultatelor se face prin apăsarea pe butonul inscripţionat **Afisare rezultate >>**

```
private void button2_Click(object sender, System.EventArgs e)
      {
            int[] vobsi=new int[cB1.Items.Count];
```



```
            for(int i=0;i<cB1.Items.Count;i++)
                  vobsi[i]=Convert.ToInt32(cB1.Items[i]);
            c.incarca_obs(vobsi);
            textBox1.Text=c.media.ToString("F");
            textBox2.Text=c.dispersia.ToString("F");
            textBox3.Text=c.coef_variatie.ToString("F");
      }
```

unde, *c* este o variabilă membră a clasei **Indicatori** de tip **Colectivitate**.

Metoda **incarca_obs()** a clasei **Colectivitate** are ca scop încărcarea observaţiilor din ComboBox în vectorul de observaţii al clasei **Colectivitate**.

La nivelul clasei **Indicatori** s-au definit şi proprietăţile pentru obţinerea valorilor indicatorilor, a numărului de observaţii şi a observaţiilor:

```
public double media
 {
      get
         {
            return c.media;
         }
 }

public int this [int i]    //acces la o observatie individuala
 {
      get
            {
                  if (i < 0 || i >= c.Nr_obs) return 0;
                  else  return c[i];
            }
 }

public int numar_obs
 {
      get
            {
                  return c.Nr_obs;
            }
 }
```

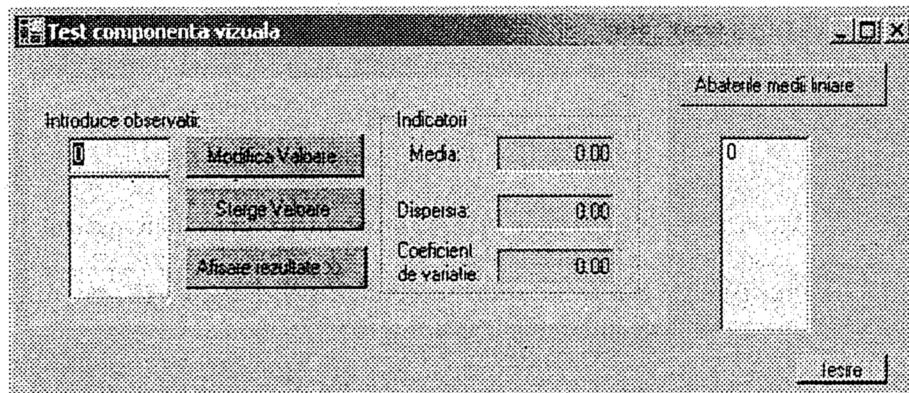
Aceste metode au ca scop mărirea flexibilităţii controlului în cazul în care acesta se integrează în alte aplicaţii.

Prin **Build / Build Solution** se obţine fişierul **comp_viz.dll** care conţine componenta cu interfaţă vizuală.

II. Utilizarea componentei cu interfață vizuală creată anterior într-o aplicație independentă presupune efectuarea următorilor pași:

1. Construirea unui nou proiect **File / New / Project** și se va alege la *Project Types* **Visual C# Project,** iar la *Templates* **Windows Application,** numele fiind `test_comp_viz`;
2. Se vizualizează fereastra *Solution Explorer* (**View / Solution Explorer**);
3. Se deschide controlul de tip *tree* care are rădăcina `test_comp_viz` și o ramură `References`;
4. Se selectează `References`, click pe butonul dreapta mouse și se alege `Add References`;
5. In casuța de dialog se selectează **Projects** după care se apasă pe butonul **Browse** și se alege fișierul `comp_viz.dll` care conține biblioteca dorită; în urma operației, la `References` va apare și `comp_viz;`
6. Se vizualizează fereastra *Toolbox*, se selectează secțiunea *General*, click pe butonul dreapta mouse și se alege opțiunea *Customize Toolbox*;
7. In căsuța de dialog *Customize Toolbox*, se selectează *.NET Frameworks Components*;
8. Se apasă butonul *Browse* și se deschide fișierul `comp_viz.dll;`
9. In controlul *ListView* va apare articolul `Indicatori` care trebuie bifat, după care se apasă butonul *OK* pentru confirmare și terminarea dialogului;
10. In secțiunea *General* a ferestrei *Toolbox* trebuie să apară articolul `Indicatori` pentru a putea draga pe formă controlul definit anterior.

Pe forma aplicației se pun controale ca în figura de mai jos.



S-au folosit:

- un control de tip `Indicatori` (instanța numită implicit `indicatori1`) în care se vor introduce observațiile statistice și se vor afișa indicatorii;
- un control de tip `ListBox` (variabila `listBox1`) pentru a afișa abaterile medii liniare ale observațiilor de la media lor. Abaterea medie liniară a unei observații (x_i) se calculează după formula:

$$x_i - m, \quad i = \overline{1, n};$$

unde: n – reprezintă numărul de observații iar m – media lor aritmetică.

La apăsarea pe butonul **Abaterile medii liniare** se preiau observațiile din controlul `indicatori1` și se afișează în controlul `listBox1` abaterile medii liniare:

```
private void button1_Click(object sender, System.EventArgs e)
 {
      double m = indicatori1.media;
      int n = indicatori1.numar_obs;
      listBox1.Items.Clear(); //sterge continutul din ListBox
      for(int i=0;i<n;i++)
            {
                  double t = indicatori1[i]-m;
                  //adaugare articol in ListBox
                  listBox1.Items.Add(t.ToString("F"));
            }
 }
```

Apăsarea pe butonul `Iesire` determină terminarea aplicației.

```
private void button2_Click(object sender, System.EventArgs e)
      {           Application.Exit();       }
```

INDEX

@ 12

A

ADO.NET 227
Alpha blending 179
arbore binar 42
arbore binar de căutare 43
ArrayList 18, 34

B

BindingContext 222
boxing 13
Button 53

C

catch 87
CheckBox 65
clasa abstractă 32
class 9, 12, 13
Clipboard 200
colecții 17
ColorDialog 145
complex data binding 223, 267
componente 298
Console 9
ContextMenu 107, 111
control de utilizator 291
Convert 59
CrystalReports 166
CrystalReportViewer 170

D

Data Grid 274
DataAdapter 232
Database Fields 168
DataBindings 219
DataRelation 241
DataRow 248
DataSet 233, 236
DataTable 233, 239
DataView 251
delegate 12, 13, 22
derivarea 30
deserializare 152
DialogBox 143
DllImport 100
drag and drop 204

E

enum 12, 13
evenimente 22, 52
excepții 87
Exit 106

F

finally 91
Font 50
FontDialog 145
for 16
foreach 16, 39
Form 49
Formula Fields 172

G

get 19
Graphics 50, 176

H

HashTable 34

I

IComparer 44
IDataObject 200, 208
IEnumerable 39
IEnumerator 39
IList 37
ImageList 114
indexare [] 74
interface 12, 13, 29

K

KeyDown 98
KeyPress 98
KeyUp 98

L

legare bidirecțională 225
legare întârziată 23
legare unidirecțională 225
lista 35
ListView 122

M

Main 9
MainMenu 107
masiv 14
masiv bidimensional 14
masiv în scară 14
MDI 148
Modal Dialog Box 143
Modless Dialog Box 144
MouseDown 51, 95
MouseMove 95
MouseUp 95
MouseWheel 95
multicasting 25
multicasting 25

N

new 11, 14, 16

O

object 34
Object 34
ODBC 227, 229
OLEDB 227
OleDbCommand 255
OnPaint 50
OpenFileDialog 145
override 69

P

Panel 176
Parameter Fields 171
PrintDocument 155
PrintPreviewDocument 161
proceduri stocate 259
proprietăți 19
proprietăți statice 20

Q

Queue 18, 34

R

RadioButon 65

S

SaveFileDialog 145
Scroll Bar 67
SDI 148
serializare 125, 152
set 19
simple data binding 219, 267
SortedList 18

Splitter	175
Stack	18, 34
StatusBar	120
stiva	41
string	11
struct	9, 12, 13

T

TextBox	60
throw	91
tipul decimal	11
tipuri de bază	9
tipuri referențiale	9
tipuri valorice	9
ToolBar	114
ToolTip	68
TreeNode	129
TreeView	128
try	87
typed DataSet	263

U

unboxing	13
untyped DataSet	263

V

validare	83
value	19
vector	14
vectori de obiecte	15
vectori de tipuri valorice	15
virtual	32
vizualizare splitată	175

W

Windows Forms	49

BIBLIOGRAFIE

1. * **, **Microsoft Developer Network Library,** Microsoft Press, 2003
2. Cappell David, **Understanding .NET, A tutorial and analysis,** Addison Wesley, San Francisco, 2002.
3. Chand Mahesh, **A Programmer's Guide to ADO.NET in C#,** Apress, New York, 2002
4. Conger David, **Programarea în C#,** Editura BIC ALL, București, 2003
5. Fox Dan **Sams Teach Yourself ADO.NET in 21 Days,** Sams Publishing, 2002
6. Joshi Bipin, DicKinson Paul, **Professional ADO.NET Programming**, Wrox Press Ltd, 2002, www.wrox.com
7. Kalani Amit, MCAD/MCSD.NET TRAINING GUIDE: **Designing and implementing Web applications with visual C# .NET and Visual Studio .NET,** QUE Publishing, 2003
8. Liberty, J., **Programming CSharp** - O'Reilly, 2003.
9. Lippman, S., **CSharp Primer - A Practical Approach -** Addison Wesley, 2002
10. Michael Stiefel, Robert J. Oberg, **Application Development Using C# and .NET,** Prentice Hall, 2001.
11. O'Brien Larry, Bruce Eckel, **Thinking in C#** , Prentice Hall, New Jersey, 2002.
12. Pappas H. Chris, Murray William, **C# pentru programarea Web,** Editura BIC ALL, București, 2004
13. Petzold Charles, **Programming Windows with C#,** Microsoft Press, 2001.
14. Pruteanu, Alexandru, **Dezvoltare de aplicații în Microsoft .NET,** note de curs, Centrul de Consultanță Microsoft, www.infoiasi.ro, 2002
15. Riordan, Rebecca M., **Microsoft ADO .NET Step by Step,** Microsoft Press, 2002
16. Schildt Herbert, **C#,** Editura Teora - Osborne, București, 2002
17. Sharp, J., Jagger, J., **Microsoft Visual C#.NET step by step,** Microsoft Press, 2003
18. Smeureanu, I., Dârdală, M., **Programarea orientată obiect în limbajul C++,** Editura CISON, București, 2002.
19. Turtschi, Adrian, Werry Jason, Hack Greg, Albahari Joseph, **C# .NET Web Developer's Guide,** Syngress Publishing, Inc., 2002
20. Wright, C., Jamsa, K., **C# programming: tips and techniques -** Osborne/McGraw-Hill/Berkeley, California, 2002